METU - EE 442 - Operating Systems

Processes and Threads

Chapter 2

Cüneyt F. Bazlamaççı (METU-EEE)

The slides include material from Tanenbaum & Bos, Modern Operating Systems, 4th ed., Pearson and Halıcı, U., EE442 Lecture Notes

Process Model (1)

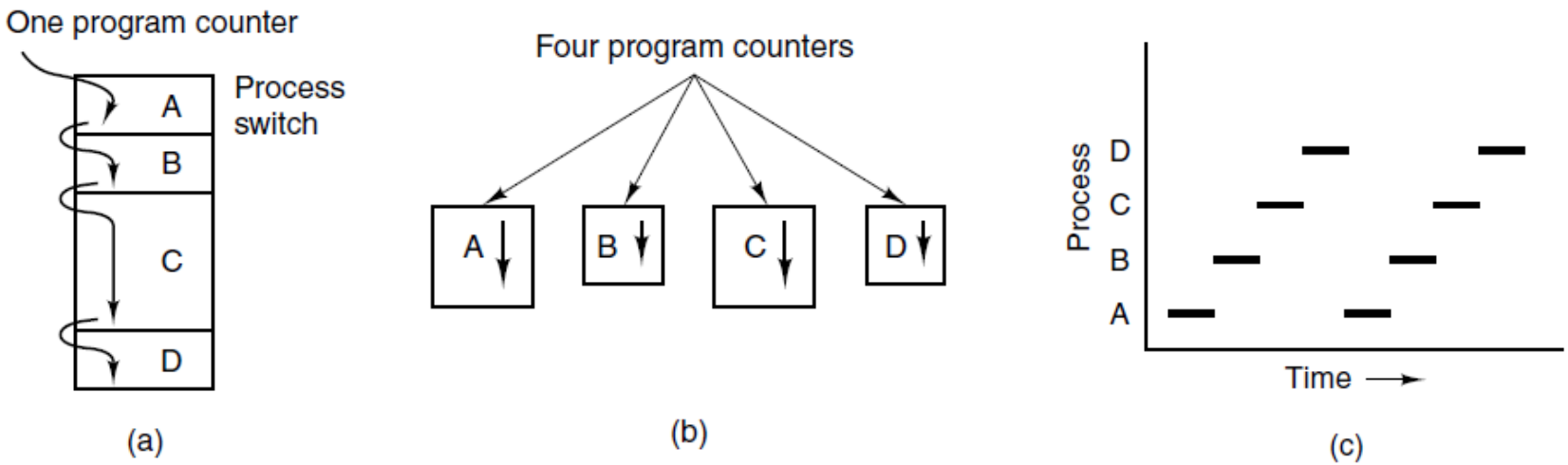
Process: is an abstraction of a running program (turns a single CPU into <u>multiple virtual CPUs</u>).

- In any **multiprogramming system**, the CPU switches from process to process quickly, running each for tens or hundreds of milliseconds.
- At any one instant the CPU is running only one process.
- In the course of 1 second it may work on several of them, giving the illusion of parallelism (**pseudoparalellism)**.

Process Model (2)

All software, including OS, is organized into a number of ***sequential processes*** (processes for short).

A **process** is an instance of an executing program, including the current values of the program counter, registers and variables.



(a) Multiprogramming of four programs.

(b) Conceptual model of four independent, sequential processes.

(c) Only one program is active at once.

Process Model (3)

- The difference between a process and a program is that the program is a group of instructions whereas the process is the activity of some kind (anology: baking scientist in text book).
  - A process has a program, input, output and a state.

- Normally, a process should not be programmed with built-in assumptions about timing.
  - When a process has critical real-time requirements, i.e., particular events must occur within a specified number of milliseconds, special measures must be taken to ensure that they do occur.

Process Creation

- Creating new processes is particularly useful when the work to be done can be formulated in terms of several related, but otherwise independent interacting processes (ex. fetch and process tasks).

- In general-purpose systems, some way exists to create and terminate processes as needed during operation.

Process Creation

Four principal events cause processes to be created:

1. System initialization.
   - Processes can run either in foreground (interacting with the user) or background.
   - Background processes (**daemons**) handle activities such as email, printing, etc. (Windows Task Manager shows them).
2. Execution of a process creation system call by a running process.
3. A user request.
4. Initiation of a batch job.

Process Creation

- After «fork», two processes, parent and the child, have
  - their own distinct address spaces
  - the same memory image,
  - the same environment strings,
  - the same open files.
- If either process changes a word in its address space, the change <u>is not</u> visible to the other process.
- In UNIX,
  - the child's initial address space is a copy of the parent's;
  - no writable memory is shared.

Process Termination

Typical conditions which terminate a process:

1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

Process Hierarchy

In UNIX, a process and all of its children and further descendants form a **process group**.

«**init**» system call is present in the boot image.

- starts running
- reads a file telling how many terminals there are.
- forks off a new process per terminal.
- each new process
  - waits for someone to log in.
  - If a login is successful, executes a shell to accept commands.

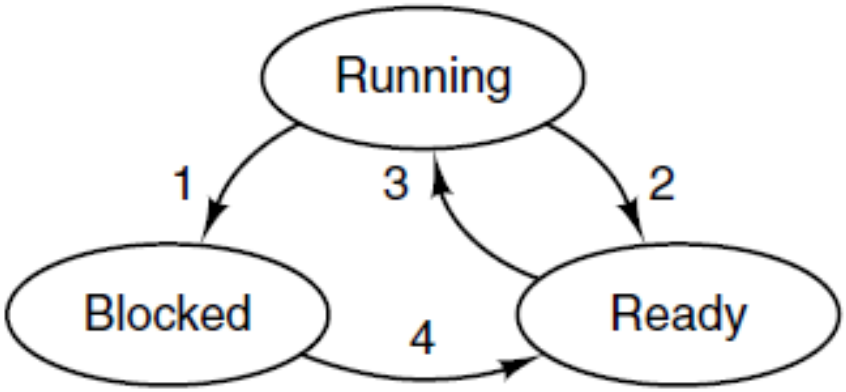
In contrast, Windows has no concept of a process hierarchy.

Process States (1)

A process may be in one of the following states:

1. Running (actually using the CPU at that instant).
2. Ready (runnable; temporarily stopped to let another process run).
3. Blocked (unable to run until some external event happens).

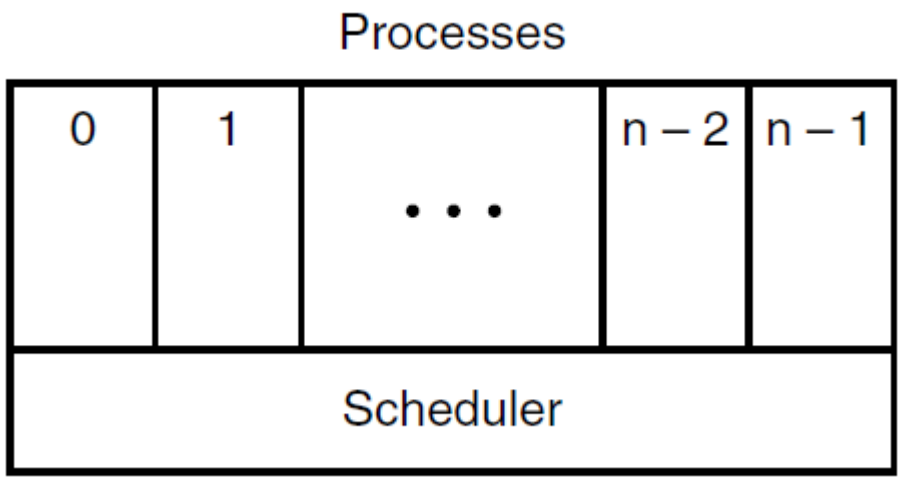
Process States (2)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

A process can be in running, blocked, or ready state.

Process States (3)



The lowest layer of a process-structured operating system handles interrupts and scheduling.

Above that layer there are sequential processes.

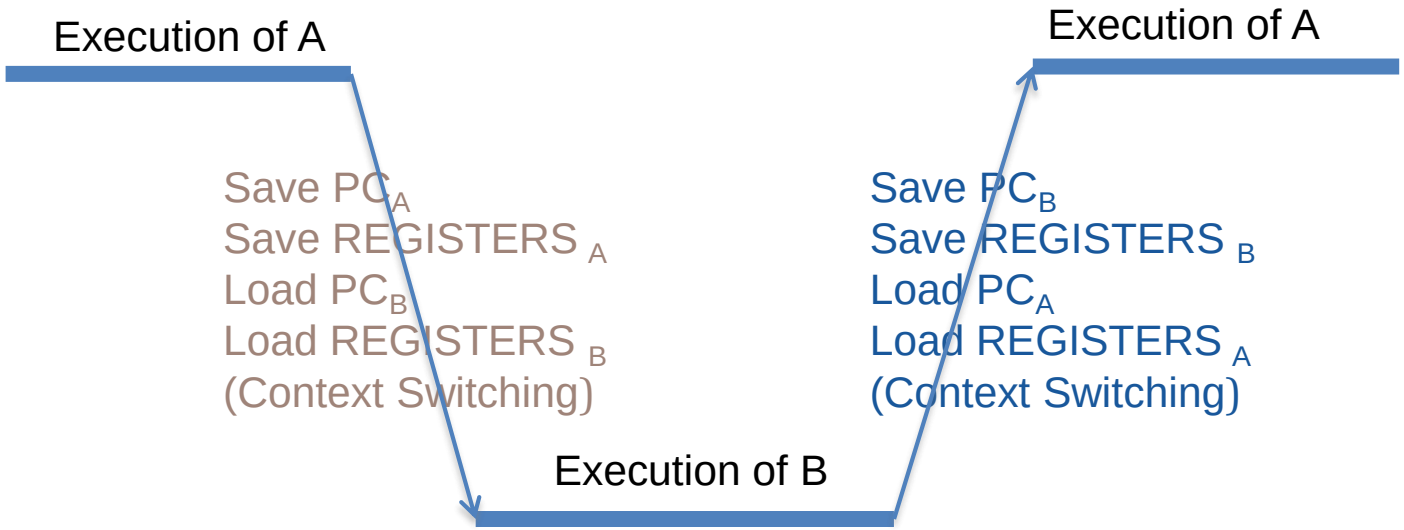
Implementation of Processes (1)

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Some of the fields of a typical process table (process control block) entry.

Implementation of Processes (2)

- Switching between processes :



Context switching during an interrupt

Implementation of Processes (3)

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Skeleton of what the lowest level of the operating system does when an interrupt occurs.

- A process may be interrupted many times during its execution and after each interrupt the interrupted process returns to <u>precisely the same state it was in</u> before the interrupt occurred.

Processes

- *Execution* of a process means CPU operations on the process such as changing its variables, performing arithmetic operations, etc.
- *I/O work* means interaction of the process with the I/O devices such as reading something or writing to somewhere.
- They may also be named as "**processor (CPU) burst**" and "**I/O burst**" respectively .

Processes

We classify programs as

- **Processor(CPU)-bound program**: A program having long processor bursts (execution instants) but shorter I/O bursts.

- **I/O-bound program**: A program having short CPU bursts but longer I/O bursts.

Multiprogramming

- Assume we have two processes A and B.
- Both execute for 1 second and do some I/O work for 1 second.
- This pattern is repeated 3 times for process A and 2 times for process B.

$Exec_{A1}$	I/O_{A1}	$Exec_{A2}$	I/O_{A2}	$Exec_{A3}$	I/O_{A3}

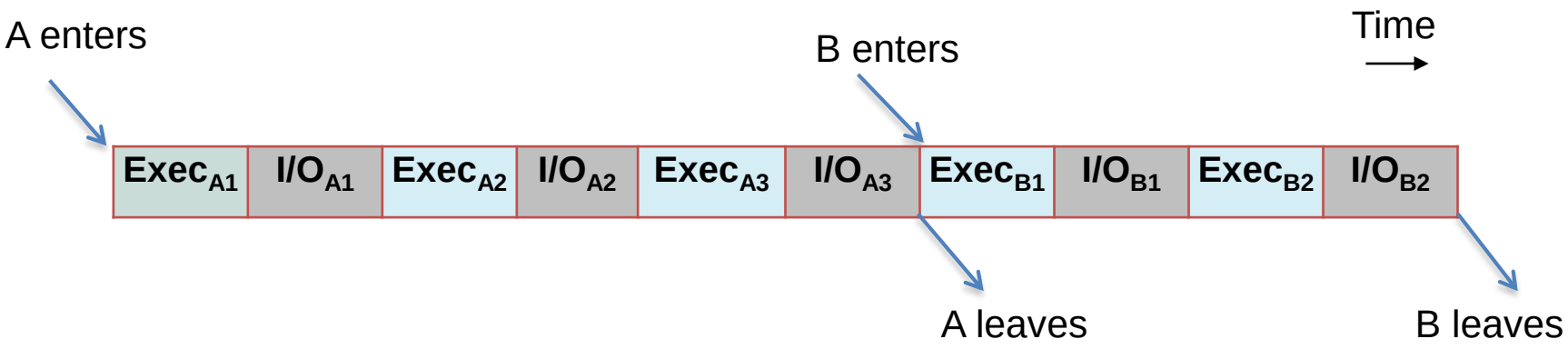
Task A

$Exec_{B1}$	I/O_{B1}	$Exec_{B2}$	I/O_{B2}

Task B

Multiprogramming

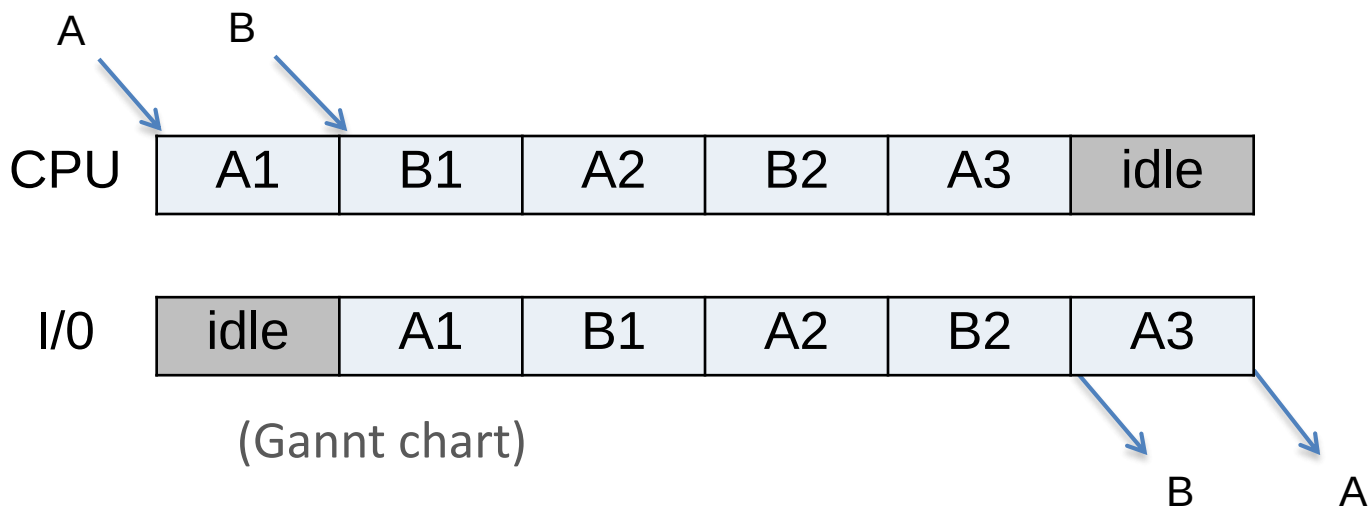
- If we have **no multiprogramming**, the processes are executed sequentially as below.



- So, the processor executes these two processes in a total time of 10 seconds.
- However, it is idle during I/O durations (idle for 5 seconds, used for 5 seconds).
- Then the processor utilization is $\frac{5}{10} \times 100 = 50\%$

Multiprogramming

Now let's consider **multiprogramming**:



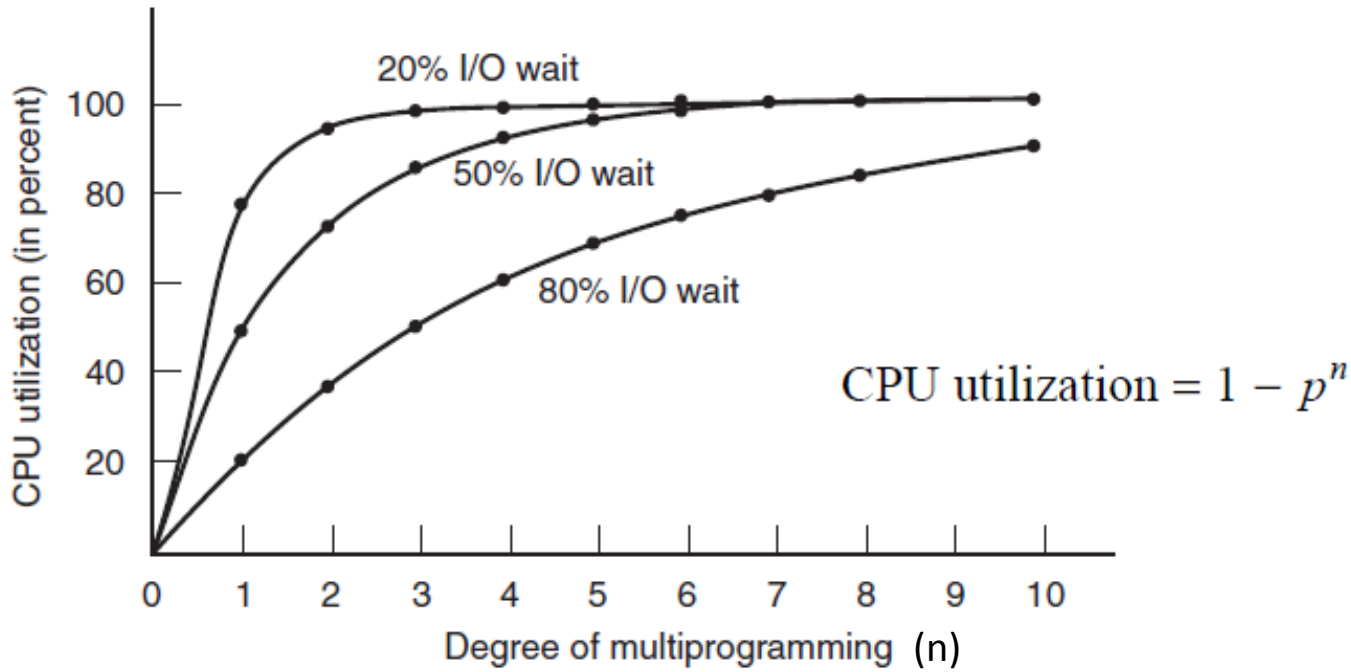
(Gannt chart)

- In this case, while process A is performing I/O work (i.e. not using the processor), CPU utilizes its time to execute process B instead of being idle.

- The processor utilization is $\frac{5}{6} \times 100 = 83\%$

Modeling Multiprogramming

- Assume that a process spends a fraction *p* of its time waiting for I/O to complete.
- With *n* independent processes in memory at once, probability that all *n* processes are waiting for I/O (meaning CPU is idle) is p^n.



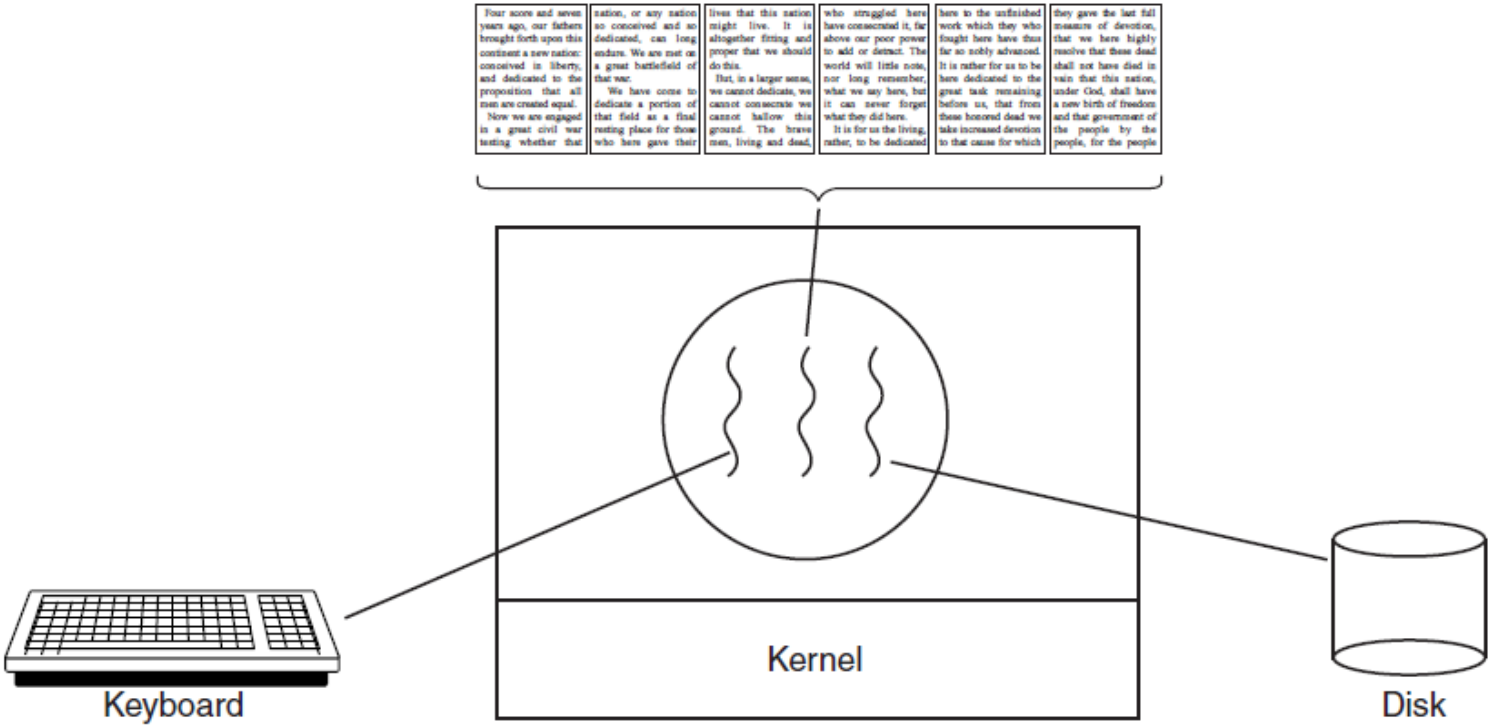
CPU utilization as a function of the number of processes in memory.

Threads

Threads are miniprocesses within a process. Threads are

- parallel entities that share an address space and all process data among themselves,
- easier (i.e., faster (maybe 10-100 times)) to create and destroy than processes,
- provide performance gain when computation and I/O are balanced,
- more useful on systems with multiple CPUs, where real parallelism exists

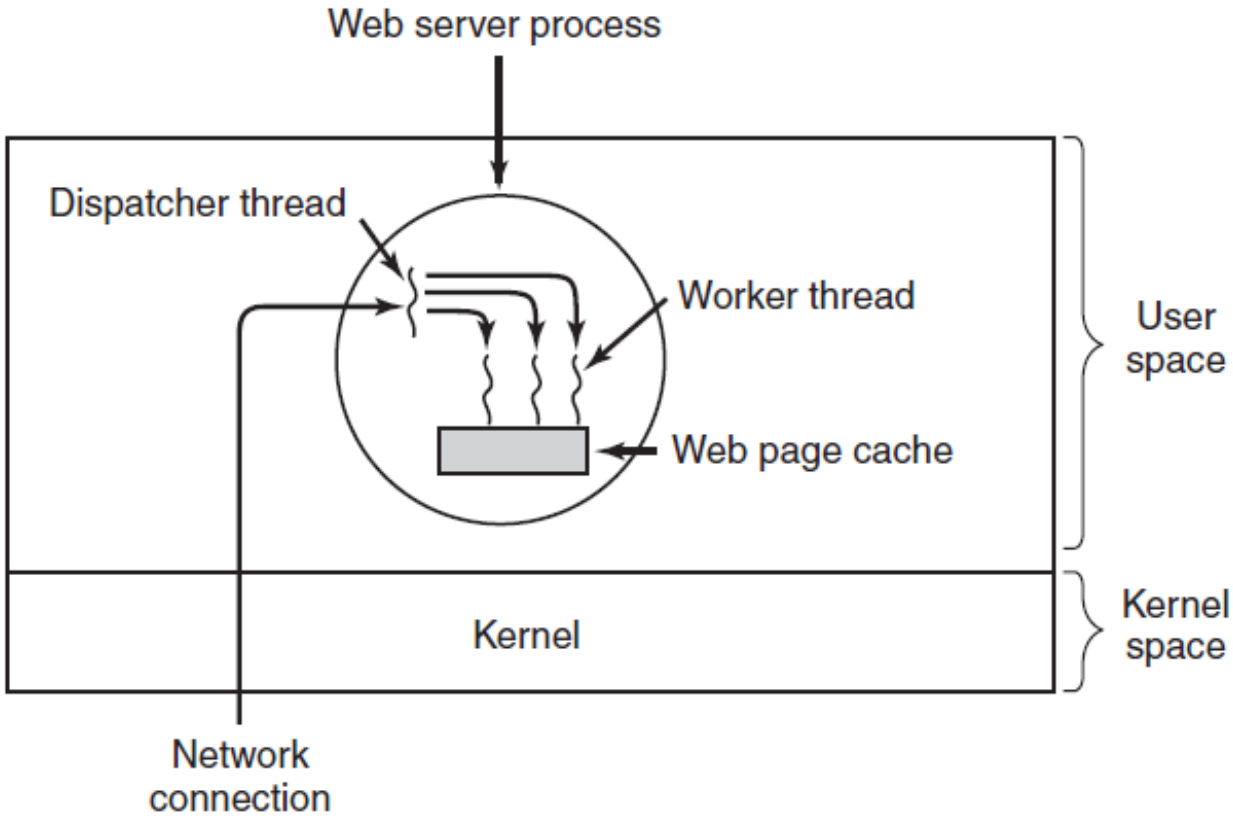
Thread Usage (1)



A word processor with three threads.

- Having three separate processes would not work here
  - all three threads need to operate on the same document (share a common memory

Thread Usage (2)



A multithreaded Web server.

Thread Usage (3)

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

A rough outline of the code for a web server.
(a) dispatcher thread. (b) worker thread.

Thread Usage (4)

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Three ways to construct a server.

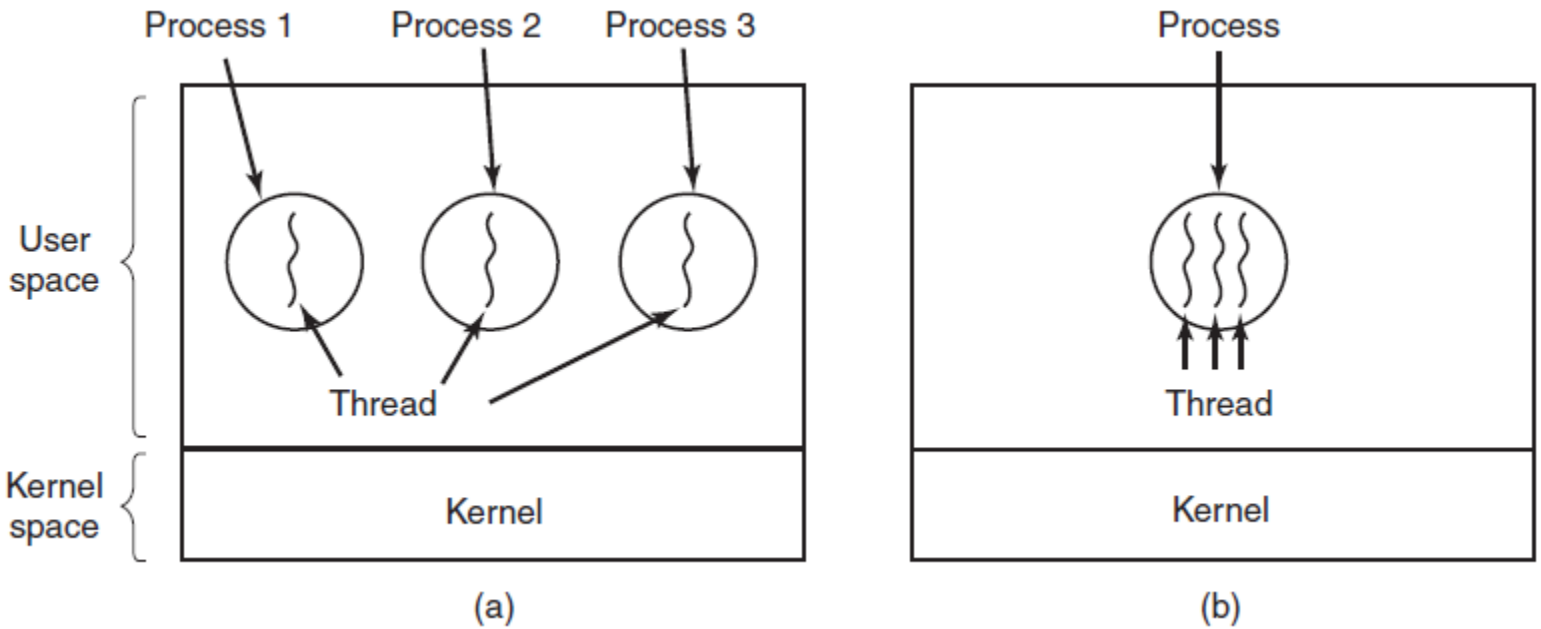
Finite-state machine style programming:

- Each computation has a saved state, and there exists some set of events (interrupts) that can occur to change the state

The Classical Thread Model (1)

Having multiple threads running in parallel in one process is analogous to having multiple processes running in parallel in one computer.

Threads are sometimes called **lightweight processes**.



(a) Three processes each with one thread (different addr. space).

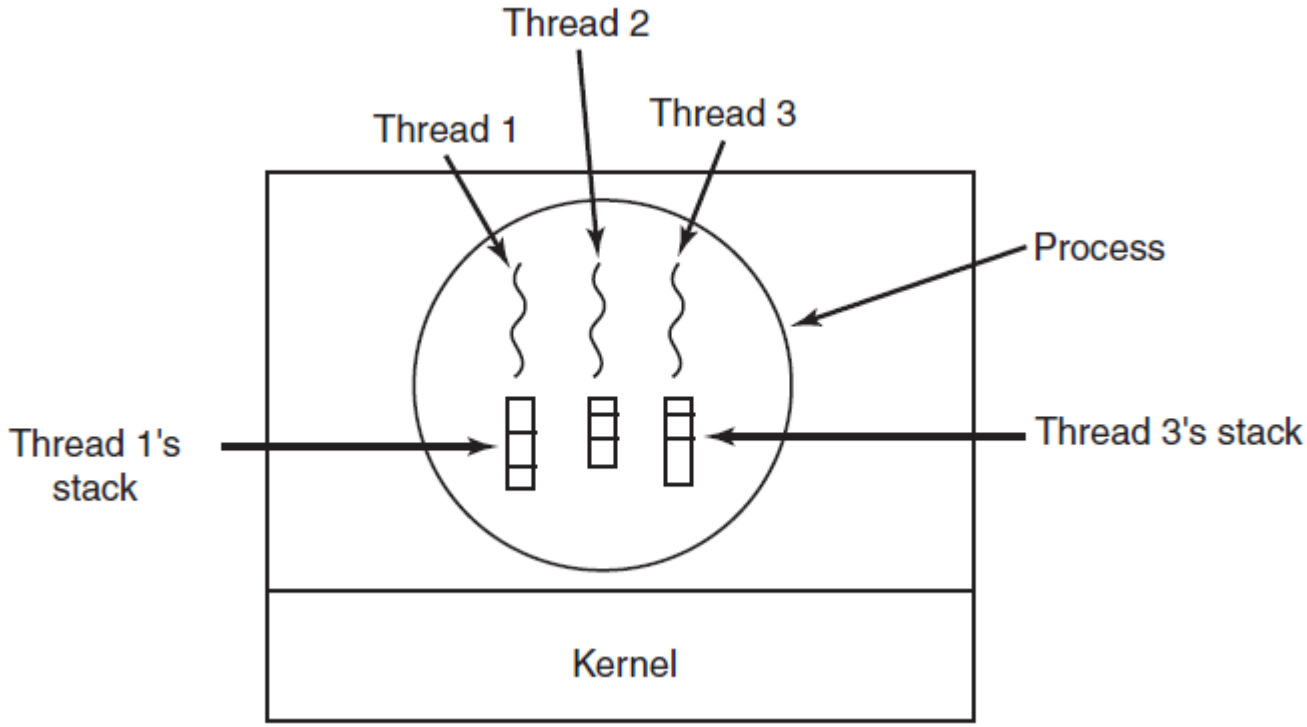
(b) One process with three threads (same addr. space) (**multithreading**)

The Classical Thread Model (2)

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

The Classical Thread Model (3)



Each thread has its own stack.

- Threads do not fight but cooperate.
- Threads are often useful but introduce complications into the programming model (about process creations, open files, new memory allocations, etc.)

POSIX Threads (1)

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Some of the Pthreads (~60 functions) function calls.

- Processes are fiercely competitive and each wants all the CPU time it can get.
- Threads of a process are working together and their code is invariably written by the same programmer.

POSIX Threads (2)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* This function prints the thread's identifier and then exits. */
        printf("Hello World. Greetings from thread %d\n", tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* The main program creates 10 threads and then exits. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d\n", i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
```

An example program using threads

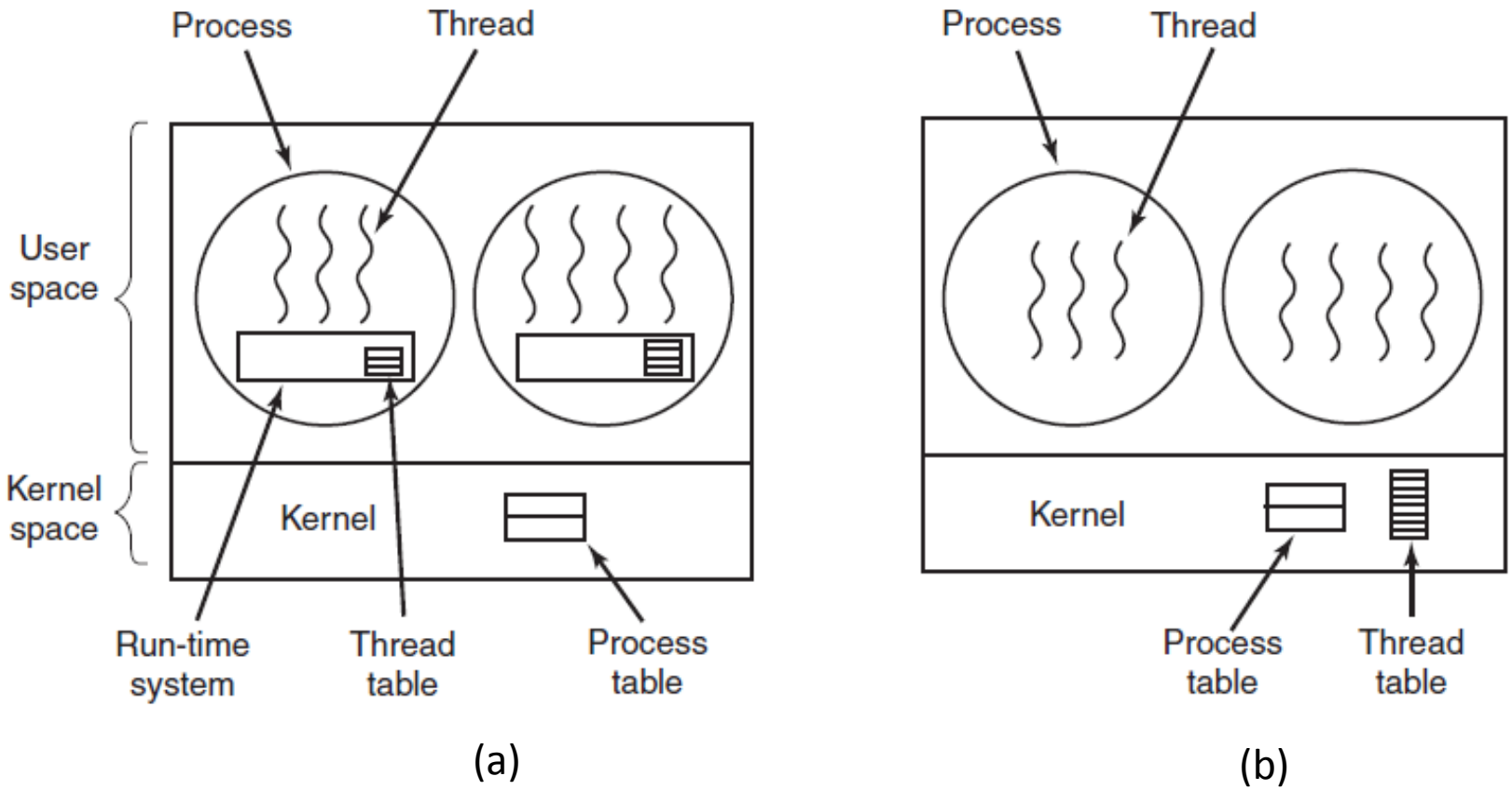
POSIX Threads (3)

```
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d\n", i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

        if (status != 0) {
            printf("Oops. pthread_create returned error code %d\n", status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

An example program using threads.

Note: The order in which various messages are interleaved is nondeterminate and may vary on consecutive runs of the program.

Implementing Threads



(a) A user-level threads package.
(b) A threads package managed by the kernel.

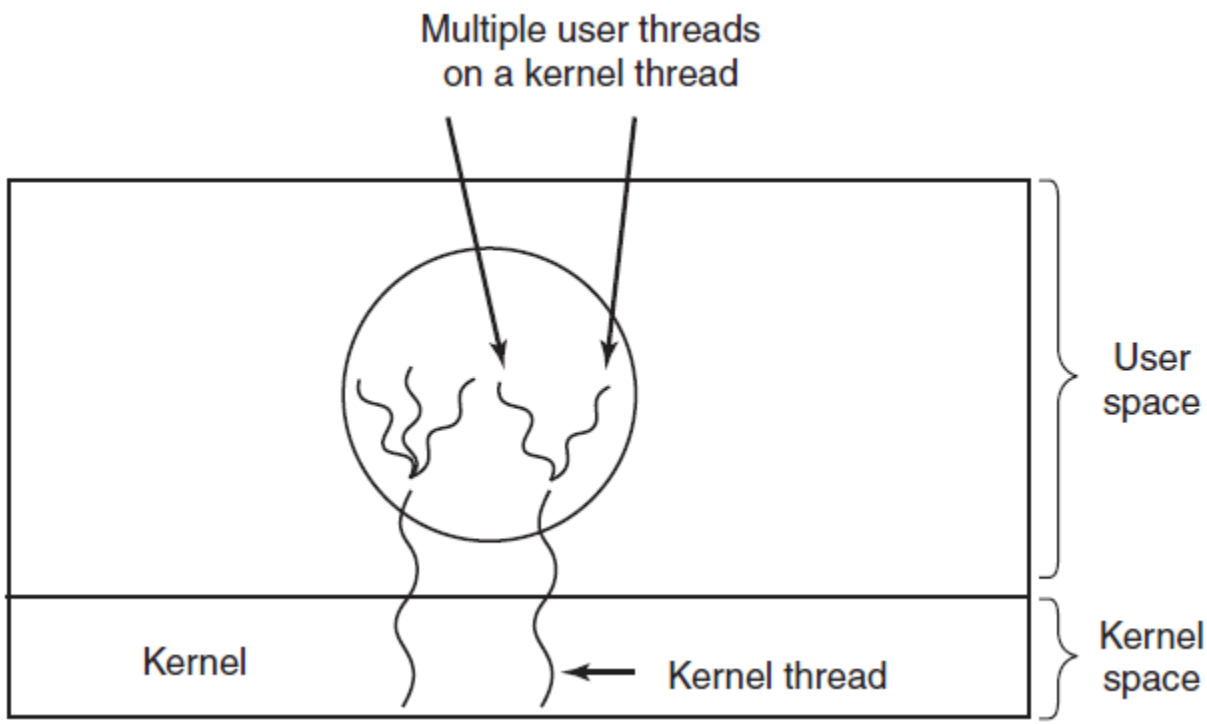
Note: pls. read pros and cons of both approaches from the text book.

Implementing Threads

- Pros of user-space thread implementation:
  - A user-level threads package can be implemented on an OS that does not support threads.
  - Doing thread switching is faster than trapping to the kernel.
  - Allow each process to have its own customized scheduling algorithm.

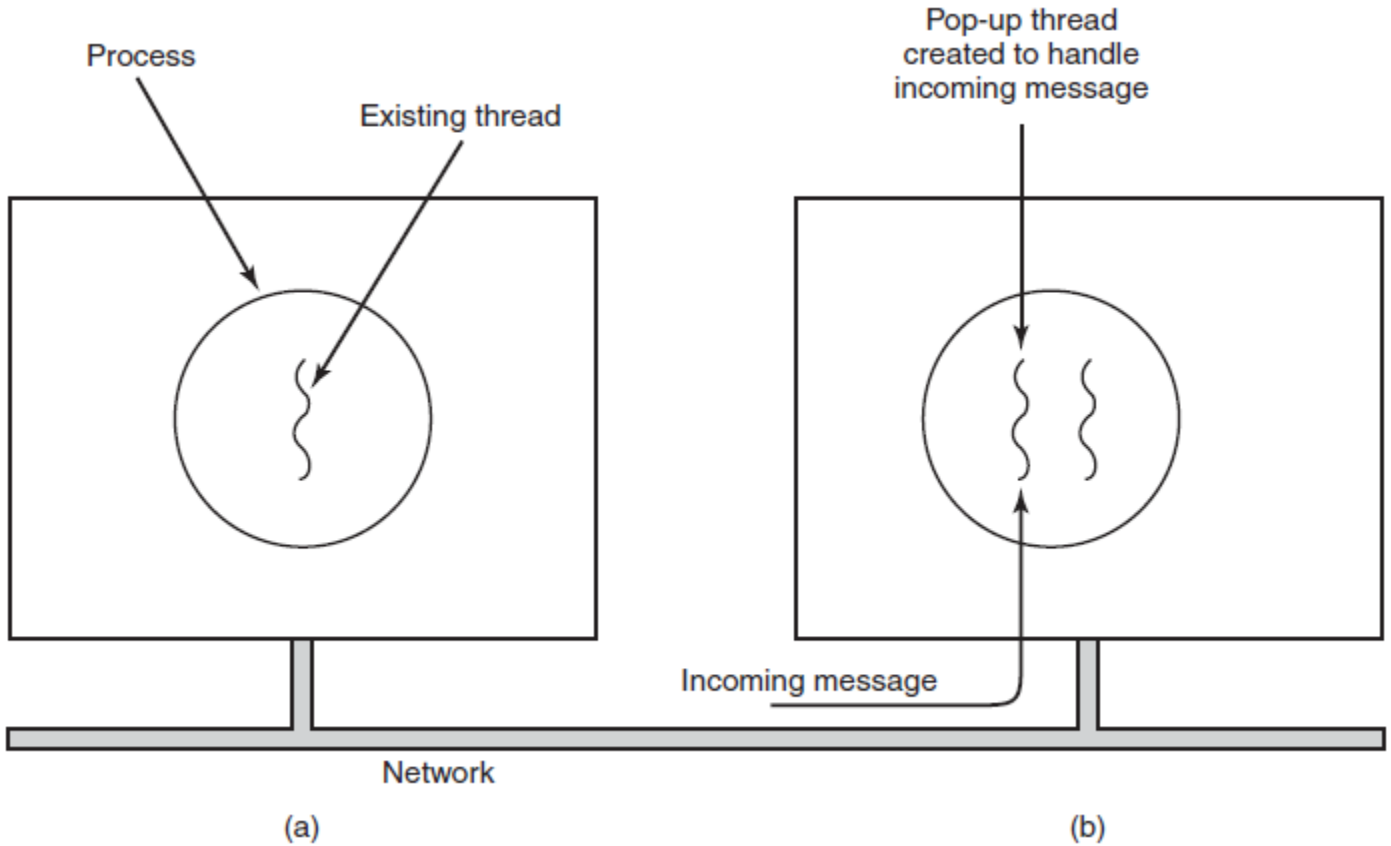
- Cons of user-space thread implementation:
  - Blocking system calls
  - Page faults
  - No clock interrupt within a process, hence scheduling difficulty.

Hybrid Thread Implementations



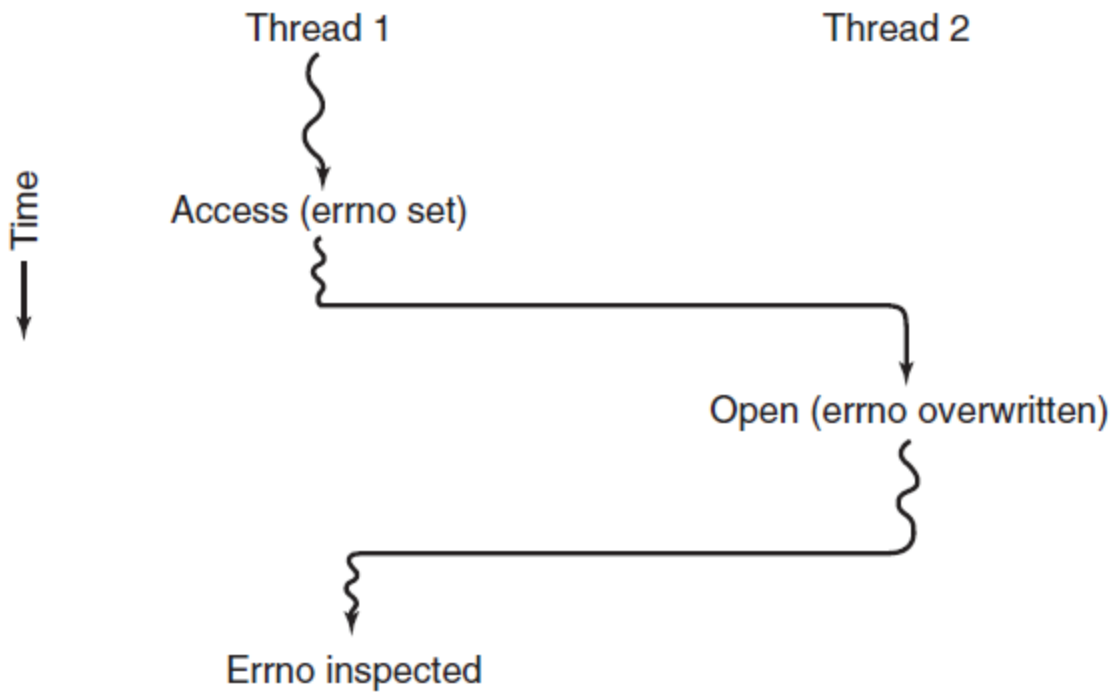
Multiplexing user-level threads onto kernel-level threads.

Pop-Up Threads



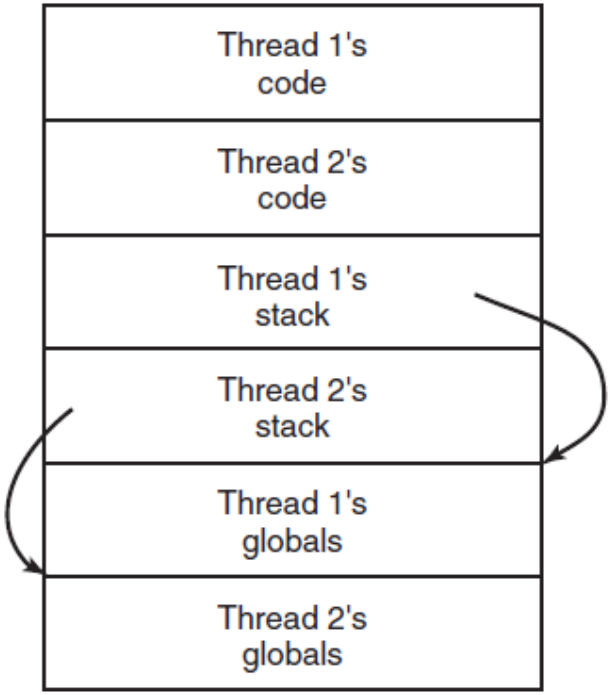
Creation of a new thread when a message arrives. (a) Before the message arrives. (b) After the message arrives.

Making Single-Threaded Code Multithreaded (1)



Conflicts between threads over the use of a global variable.

Making Single-Threaded Code Multithreaded (2)



Threads can have private global variables.

- This requires a new scoping level, i.e., variables visible to all the procedures of a thread (but not to other threads), in addition to the existing scoping levels of variables.

Making Single-Threaded Code Multithreaded (3)

- Many existing programs are single-threaded processes.
- Converting these to multithreading is trickier.
- Issues to be handled include:
  - use of global variables
  - accessing private global variables if used
  - not reentrant library procedures
  - use of dynamic memory allocation procedures

Interprocess Communication (IPC)

- There is a need for communication between processes, preferably in a well-structured way not using interrupts.

- Three brief concerns are :
  - How can one process pass information to another?
  - How to make sure that two or more processes do not get in each other's way?
  - How to do proper sequencing when dependencies are present?

- Processes and threads have similar IPC problems and solutions (maybe except the first item).

Interprocess Communication

- Ex: Consider a system such that
  - When a process wants to print a file,
    - it enters the file name in a special *spooler directory*.
  - Another process, the *printer daemon*,
    - periodically checks if there is any file to be printed,
    - if yes, sends it to printer and removes its name from the directory.

Interprocess Communication

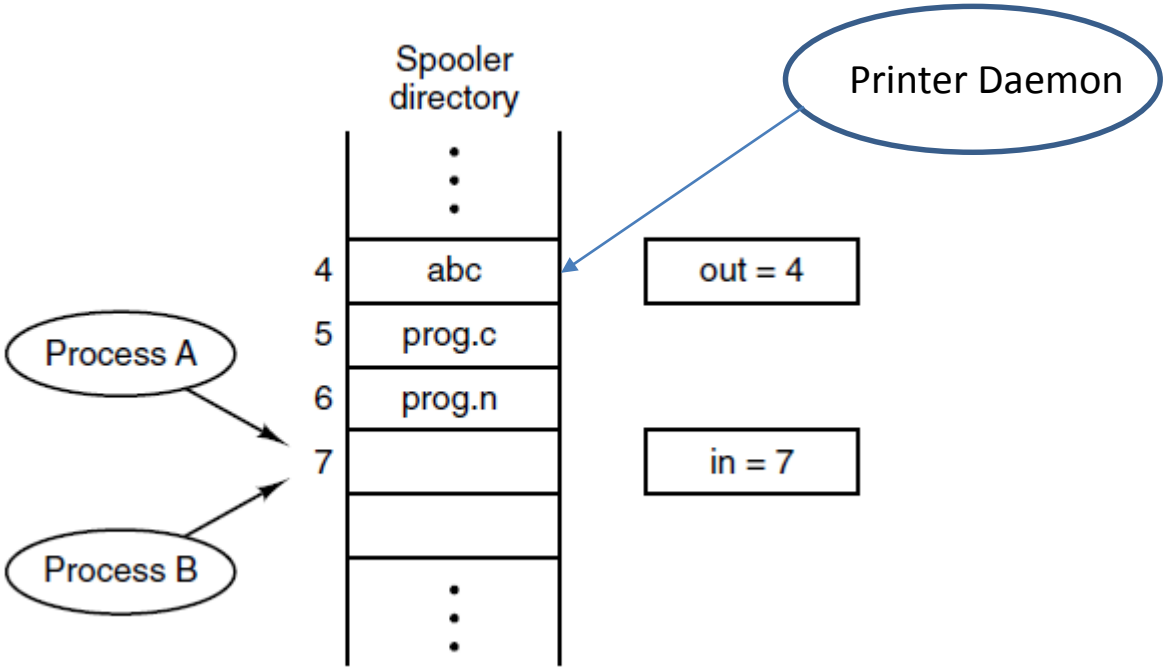
- Assume that spooler directory has a large number of slots (0, 1, 2, …,) each capable of holding a file name.

- Also suppose two shared variables exist:
  - out: points in the spooler to the next file entry to be printed,
  - in: points to the next free slot to enter a new filename in the spooler directory.

- These variables are available to all processes.

Race Conditions

Situations, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called **race conditions**.



Two processes want to access shared memory at the same time.

Critical Section Problem

Common integer variables on files in and out
global variable: spooler [n]

Process A:	Process B:	Printer Deamon:

{local variables:
char *filename_a,
int a_in;
while (TRUE) {
prepare_file(filename_a);
read(in, a_in); // a_in=7
spooler[a_in]=filename_a
A_in=(a_in+1) % n
write(in, a_in) // in=8
}
}
	{local variables:
	char *filename_b,
	İnt b_in;
	while (TRUE) {
	prepare_file(filename_b);
	read(in, b_in); // b_in=7
	spooler[b_in]=filename_b
	b_in=(b_in+1) % n
	write(in, b_in) // in=8
	}
	}
		{
		while (TRUE) {
		read(out, p_out);
		filename=spooler[p_out]
		print(filename)
		p_out=p_out+1 % n
		write (out, p_out);
		}
		}

In the above scenario, the spooler directory is internally consistent, printer daemon will not notice anything wrong, but process B will not get its output.

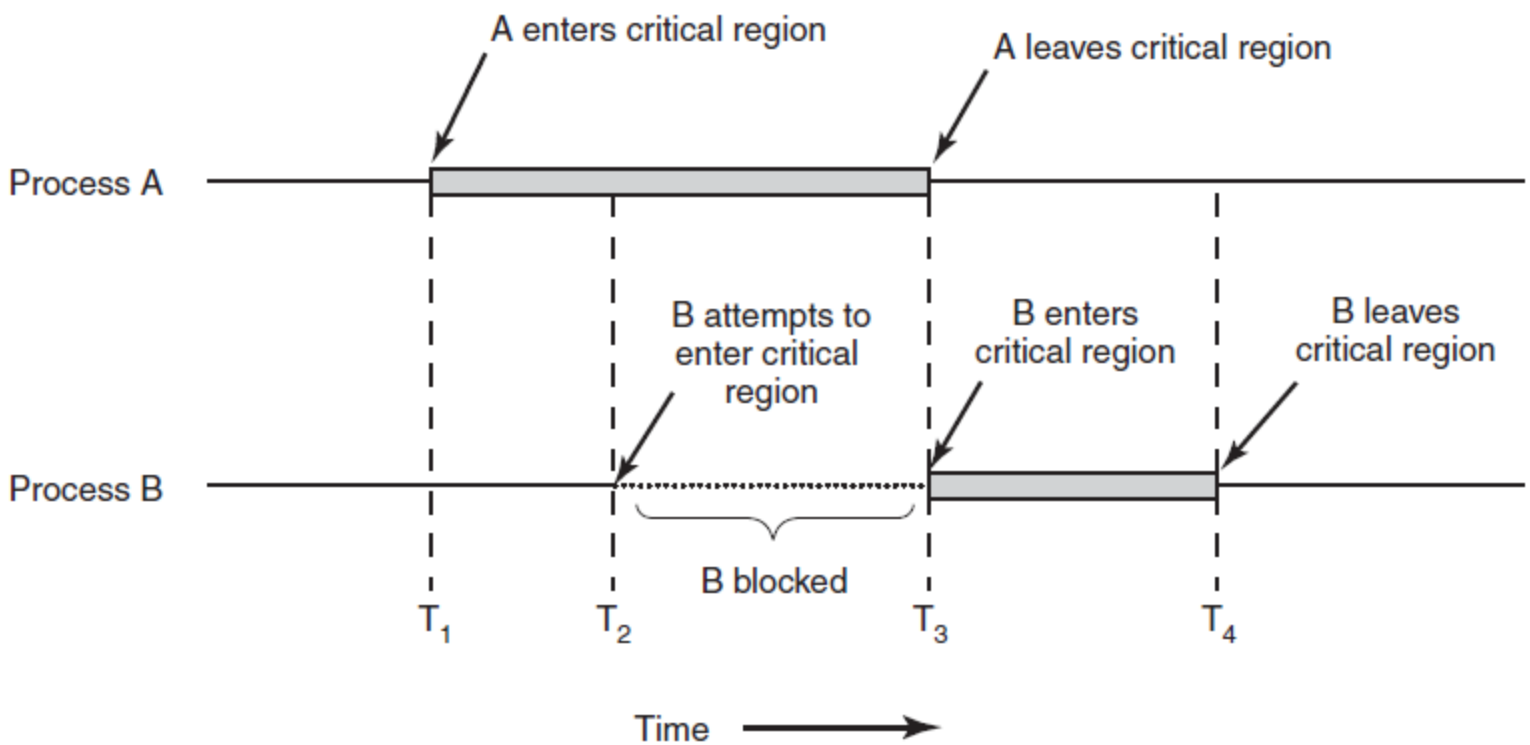
Critical Regions (1)

- How do we avoid race conditions?
  - Answer: use **mutual exclusion** (make sure that if one process is using a shared variable or file or anything (i.e. it is in its **critical region**), other processes are excluded from doing the same thing)

Requirements to avoid race conditions:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

Critical Regions (2)



Mutual exclusion using critical regions.

Critical Section Problem

- Consider two process p0 and p1 in the form:

{common variable declarations and initializations}	

P0:
{
while (TRUE) {
CS( ) ;
Non-CS( ) ;
}
}
	P1:
	{
	while (TRUE) {
	CS( ) ;
	Non-CS( ) ;
	}
	}

Critical Section Problem

- Consider two process p0 and p1 in the form:

{common variable declarations and initializations}	

P0:
{
while (TRUE) {
{CS entry code}
CS( ) ;
{CS exit code}
Non-CS( ) ;
}
}
	P1:
	{
	while (TRUE) {
	{CS entry code}
	CS( ) ;
	{CS exit code}
	Non-CS( ) ;
	}
	}

Disabling Interrupts

- The simplest solution for CS:
  - each process disables all interrupts just after entering its CS
  - re-enables them just before leaving it.
- With interrupts disabled, processor cannot switch to another process
  - thus, it can examine and update the shared memory without fear that another process will intervene.
- This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts.
- Suppose one of them did it, and never turned them on again.
- This may be the end of the system.

Lock Variables (software solution)

- As a second attempt, let us look for a software solution.
  - Consider having a single, shared lock variable initialized to 0.
  - When a process wants to enter its CS, it first tests the lock.
    - If the lock is 0, the process sets it to 1 and enters the CS
    - If the lock is already 1, the process just waits until it becomes 0.

Lock Variables (software solution)

```
#define FALSE 0
#define TRUE   1
int lock=FALSE
```

{CS entry code}

{CS exit code}

```
PO:
{
  while (TRUE) {
    while (lock) { }; /* wait */
    lock=TRUE;
    CS( ) ;
    lock=FALSE;
    Non-CS( ) ;
  }
}
```

```
P1:
{
  while (TRUE) {
    while (lock) { }; /* wait */
    lock=TRUE;
    CS( ) ;
    lock=FALSE;
    Non-CS( ) ;
  }
}
```

- Unfortunately, this idea contains a fatal flaw.

Lock Variables (software solution)

Consider the following execution order

int lock=FALSE	

PO:
{
1 while (TRUE) {
2 while (lock) { }; /* wait */
5 lock=TRUE;
6 CS( ) ;
lock=FALSE;
Non-CS( ) ;
}
}
	P1:
	{
	3 while (TRUE) {
	4 while (lock) { }; /* wait */
	7 lock=TRUE;
	8 CS( ) ;
	lock=FALSE;
	Non-CS( ) ;
	}
	}

Both processes enters the CS !!!

Lock Variables (software solution)

To overcome problem, change the code as follows:

int lock=FALSE	

PO:
{
while (TRUE) {
lock=TRUE;
while (lock) { }; /* wait */
CS( ) ;
lock=FALSE;
Non-CS( ) ;
}
}
	P1:
	{
	while (TRUE) {
	lock=TRUE;
	while (lock) { }; /* wait */
	CS( ) ;
	lock=FALSE;
	Non-CS( ) ;
	}
	}

This time, we have another problem…

Lock Variables (software solution)

Consider the following execution order:

int lock=FALSE	

PO:
{
1 while (TRUE) {
2 lock=TRUE;
3 while (lock) { }; /* wait */
CS( ) ;
lock=FALSE;
Non-CS( ) ;
}
}
	P1:
	{
	4 while (TRUE) {
	5 lock=TRUE;
	6 while (lock) { }; /* wait */
	CS( ) ;
	lock=FALSE;
	Non-CS( ) ;
	}
	}

Both processes wait forever !!! (**deadlocked** !)

Mutual Exclusion with Busy Waiting: Strict Alternation (software solution)

```
while (TRUE) {
    while (turn != 0);        /* loop */
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

```
while (TRUE) {
    while (turn != 1);        /* loop */
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b)

A proposed solution to the critical region problem (turn=0 initially).

(a) Process 0 (b) Process 1

- Note: Taking turns is not a good idea when one of the processes is much slower than the other (violates condition 3)

Mutual Exclusion with Busy Waiting: Peterson's Solution (software solution)

```
#define FALSE  0
#define TRUE   1
#define N      2                          /* number of processes */

int turn;                                 /* whose turn is it? */
int interested[N];                        /* all values initially 0 (FALSE) */

void enter_region(int process);           /* process is 0 or 1 */
{
        int other;                        /* number of the other process */

        other = 1 – process;              /* the opposite of process */
        interested[process] = TRUE;       /* show that you are interested */
        turn = process;                   /* set flag */
        while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)            /* process: who is leaving */
{
        interested[process] = FALSE;      /* indicate departure from critical region */
}
```

Peterson's solution for achieving mutual exclusion.

Special Hardware Instructions

- <u>Test-and-set</u>

- The instruction test-and-set(int lock)
  - returns true if lock is true,
  - otherwise it first sets the value of lock to true and returns false.

- It can be used for the solution of the CS problem as follows:

```
int  lock=FALSE;
```

PO:	P1:
{	{
while (TRUE) {	while (TRUE) {
while test-and-set(lock) { } ;	while test-and-set(lock) { };
CS;	CS;
lock = FALSE;	lock = FALSE;
non-CS;	non-CS;
}	}
}	}

Mutual Exclusion with Busy Waiting: The TSL Instruction (1) (h/w solution)

```
enter_region:
        TSL REGISTER,LOCK        | copy lock to register and set lock to 1
        CMP REGISTER,#0          | was lock zero?
        JNE enter_region         | if it was nonzero, lock was set, so loop
        RET                      | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0             | store a 0 in lock
        RET                      | return to caller
```

Entering and leaving a critical region using the TSL instruction.

Special Hardware Instructions

- <u>Swap</u>

- The instruction swap (int lock, int key) interchanges the content of its parameters.

int lock=0;	

PO:
{ int key
while (TRUE) {
key=TRUE
while (KEY) swap (lock,key);
CS;
lock = FALSE;
non-CS;
}
}
	P1:
	{ int key
	while (TRUE) {
	key=TRUE
	while (KEY) swap (lock,key);
	CS;
	lock = FALSE;
	non-CS;
	}
	}

Mutual Exclusion with Busy Waiting: The TSL Instruction (2) (h/w solution)

```
enter_region:
        MOVE REGISTER,#1            | put a 1 in the register
        XCHG REGISTER,LOCK          | swap the contents of the register and lock variable
        CMP REGISTER,#0             | was lock zero?
        JNE enter_region            | if it was non zero, lock was set, so loop
        RET                         | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0                | store a 0 in lock
        RET                         | return to caller
```

Entering and leaving a critical region using the XCHG instruction

Peterson, TSL and XCHG solutions are still problematic (ex. priority inversion)

Sleep and Wakeup

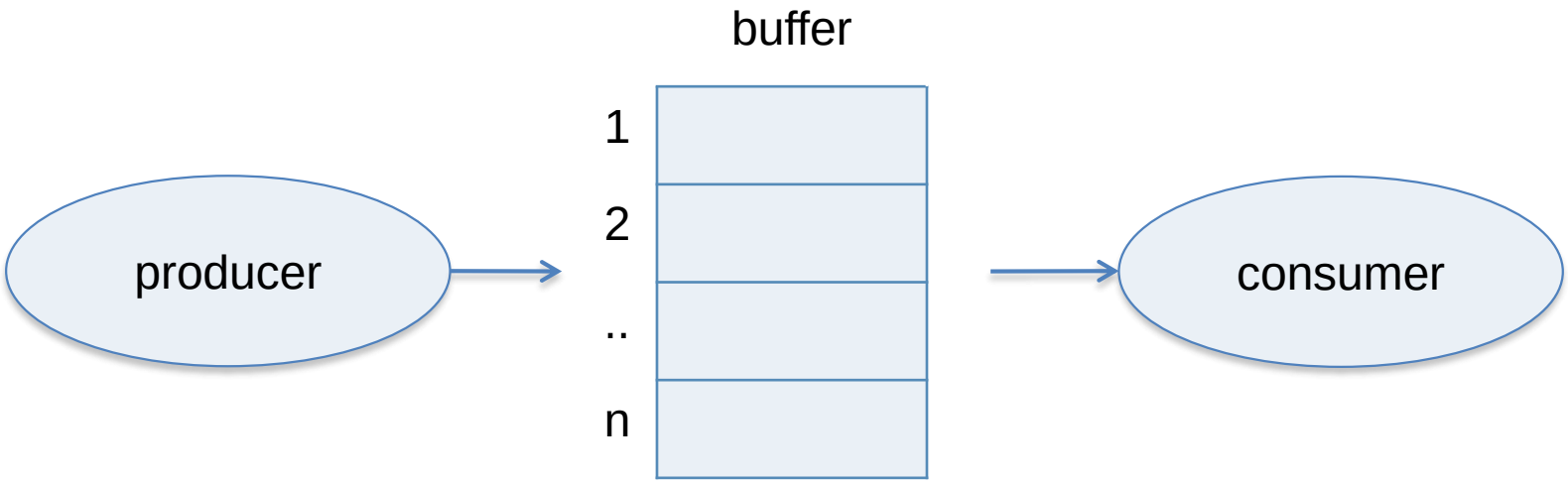
IPC primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions exist:

- *sleep* system call causes the caller to block, that is, be suspended until another process wakes it up.
- *wakeup* call has one parameter, namely the process to be awakened.

- Alternatively, both sleep and wakeup each have one parameter, a memory address used to match up sleeps with wakeups.

Bounded Buffer (Producer-Consumer) Problem (1)

- Assume a common buffer having n slots capable of holding one item.
- Process producer will produce items to fill slots.
- Process consumer will consume the items in these slots.



- The consumer should not try to consume items which have not been produced yet (i.e. the consumer can not consume empty slots).
- The producer should not try to put item into filled slots.

Sleep and Wakeup
The Producer-Consumer Problem (2)

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                item = produce_item( );         /* generate next item */
                if (count == N) sleep( );       /* if buffer is full, go to sleep */
                insert_item(item);              /* put item in buffer */
                count = count + 1;              /* increment count of items in buffer */
                if (count == 1) wakeup(consumer); /* was buffer empty? */
        }
}


void consumer(void)
{
```

The producer-consumer problem with a fatal race condition.

Sleep and Wakeup
The Producer-Consumer Problem (3)

```
        if (count == 1) wakeup(consumer);    /* was buffer empty? */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                                  /* repeat forever */
                if (count == 0) sleep( );               /* if buffer is empty, got to sleep */
                item = remove_item( );                  /* take item out of buffer */
                count = count – 1;                      /* decrement count of items in buffer */
                if (count == N – 1) wakeup(producer);   /* was buffer full? */
                consume_item(item);                     /* print item */
        }
}
```

The producer-consumer problem with a fatal race condition.

Sleep and Wakeup
The Producer-Consumer Problem (4)

- *count* variable is shared hence similar problems as in print spooler case may be encountered.

- Assume the following scenario:
  - Consumer is running.
  - The buffer is empty.
  - Consumer has just read *count* to see if it is 0.
  - Scheduler stops running the consumer and starts the producer.
  - Producer inserts an item in the buffer, increments *count*, notices it is now 1.
  - Producer calls wakeup to wake the consumer up.
  - Consumer is not logically asleep yet, so wakeup signal is lost.
  - When consumer runs next, it will test the value of count it previously read, finds it to be 0, and go to sleep.
  - Sooner or later the producer will fill up the buffer and also go to sleep.
  - Both will sleep forever.

- The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost.

Semaphores

- For IPC problems, E. W. Dijkstra proposed, a new variable type, called **semaphore** together with *two atomic operations*, DOWN and UP.

- The DOWN operation on a semaphore
  - checks to see if the value is greater than 0.
  - If so, it decrements the value and just continues.
  - If the value is 0, the process is blocked (or *put to sleep)*.

- The UP operation on a semaphore
  - increments the value of the semaphore addressed
  - *if it becomes 1, the process wakes a sleeping process up and allows it to decrement semaphore*.

- Checking the value, changing it, and possibly going to sleep are all done as a single, indivisible action (hence are called as *atomic* actions).

- It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or is blocked.

Semaphores (for CS problem)

```
typedef int semaphore;   /* semaphores are a special kind of int */
semaphore s=0;
```

```
void down(semaphore &s)
{ while (s <= 0) { } ;
  s = s – 1;
}
```

```
void up(semaphore &s)
{ s=s+1;
}
```

Note that this is blocking type.

- If two processes try to execute up(s) or down(s) simultaneously, these operations will be executed sequentially in an arbitrary order.

```
semaphore mutex = 1;    /* controls access to CS */
```

```
proces_i (void)                    /* n-th process*/
 {   while (TRUE) {
        down(mutex) ;              /* CS_entry */
        CS
        up (mutex) ;               /* CS_exit */
        non-CS
     }
   }
```

Use of semaphores for CS problem

Producer-Consumer Problem using Semaphores (1)

```
#define N 100                          /* number of slots in the buffer */
typedef int semaphore;                 /* semaphores are a special kind of int */
semaphore mutex = 1;                   /* controls access to critical region */
semaphore empty = N;                   /* counts empty buffer slots */
semaphore full = 0;                    /* counts full buffer slots */

void producer(void)
{
        int item;

        while (TRUE) {                 /* TRUE is the constant 1 */
                item = produce_item( );/* generate something to put in buffer */
                down(&empty);          /* decrement empty count */
                down(&mutex);          /* enter critical region */
                insert_item(item);     /* put new item in buffer */
                up(&mutex);            /* leave critical region */
                up(&full);             /* increment count of full slots */
        }
}

void consumer(void)
```

Producer-Consumer Problem using Semaphores (2)

```
            up(&full);                          /* increment count of full slots */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* infinite loop */
            down(&full);                        /* decrement full count */
            down(&mutex);                       /* enter critical region */
            item = remove_item( );              /* take item from buffer */
            up(&mutex);                         /* leave critical region */
            up(&empty);                         /* increment count of empty slots */
            consume_item(item);                 /* do something with the item */
        }
}
```

Semaphores (for CS problem)

- Be sure you understand that
  - using TSL or XCHG to prevent several CPUs from accessing the semaphore at the same time
  - producer or consumer busy waiting for the other to empty or fill the buffer

  are quite different.

- The semaphore operation will take only a few microseconds, whereas the producer or consumer might take arbitrarily long.

Semaphores (for process sycnhronization)

- Ex: Consider two concurrent processes: p1, p2.
  - Statement st2 of p2 is required to be executed after statement st1 of p1.

void P1(void) { … st1 .... }	void P2(void) { ... st2 ... }

Semaphores (for process sycnhronization)

- Ex: Consider two concurrent processes: p1, p2.
  - Statement st2 of p2 is required to be executed after statement st1 of p1.

Semaphore synch=0;	

void P1(void)
{ …
st1;
up(synch);
....
}
	void P2(void)
	{ ...
	down(synch);
	st2;
	...
	}

Another Synch Example

- Example: Consider two concurrent processes P1 and P2
  - P1 and P2 are executing st1 and st2 respectively in an infinite loop
  - Statements st1 and st2 are reqired to be executed exactly as follows:

    st1 st2 [st1 st2] repeated

void P1(void) { while (TRUE) { .... st1; .... } }	void P2(void) {while (TRUE) { .... st2; .... } }

Another Synch Example

- Example: Consider two concurrent processes P1 and P2
  - P1 and P2 are executing st1 and st2 respectively in an infinite loop
  - Statements st1 and st2 are reqired to be executed exactly as follows:

    st1 st2 [st1 st2] repeated

Semaphore synch12=0, synch21=1;	

void P1(void)
{ while (TRUE)
{ ....
down(synch21);
st1
up(synch12);
....
}
}
	void P2(void)
	{while (TRUE)
	{ ...
	down(synch12);
	st2
	up(synch21);
	...
	}
	}

Mutexes

A **mutex**

- is a shared variable that can be in one of two states: unlocked or locked and
- can easily be implemented in user space provided that a TSL or XCHG instruction is available.

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Implementation of *mutex_lock* and *mutex_unlock*.

Mutexes in Pthreads (1)

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Some of the Pthreads calls relating to mutexes.

Mutexes in Pthreads (2)

In addition to mutexes, Pthreads offers a second synchronization mechanism: condition variables.

- Mutexes are good for allowing or blocking access to a critical region.
- Condition variables allow threads to block due to some condition not being met.

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Some of the Pthreads calls relating to condition variables.

Mutexes in Pthreads (3)

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                    /* used for signaling */
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);  /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                      /* put item in buffer */
                pthread_cond_signal(&condc);     /* wake up consumer */
                pthread_mutex_unlock(&the_mutex); /* release access to buffer */
        }
        pthread_exit(0);
}
```

Using threads to solve the producer-consumer problem.

Mutexes in Pthreads (4)

```
        pthread_exit(0);
}

void *consumer(void *ptr)                          /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);    /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                        /* take item out of buffer */
                pthread_cond_signal(&condp);       /* wake up producer */
                pthread_mutex_unlock(&the_mutex); /* release access to buffer */
        }
        pthread_exit(0);
}

int main(int argc, char **argv)
```

Using threads to solve the producer-consumer problem.

Mutexes in Pthreads (5)

```
        pthread_exit(0);
}

int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

Using threads to solve the producer-consumer problem.

Shared Space among Processes

- There has been an assumption that multiple processes have access to at least some shared memory.
- If processes have disjoint address spaces, how can this be achieved?
- There are two answers.
  - First, shared data structures, such as semaphores, can be stored in the kernel and accessed only by means of system calls.
  - Second, most modern OSs (including UNIX and Windows) offer a way for processes to share some portion of their address space with other processes.

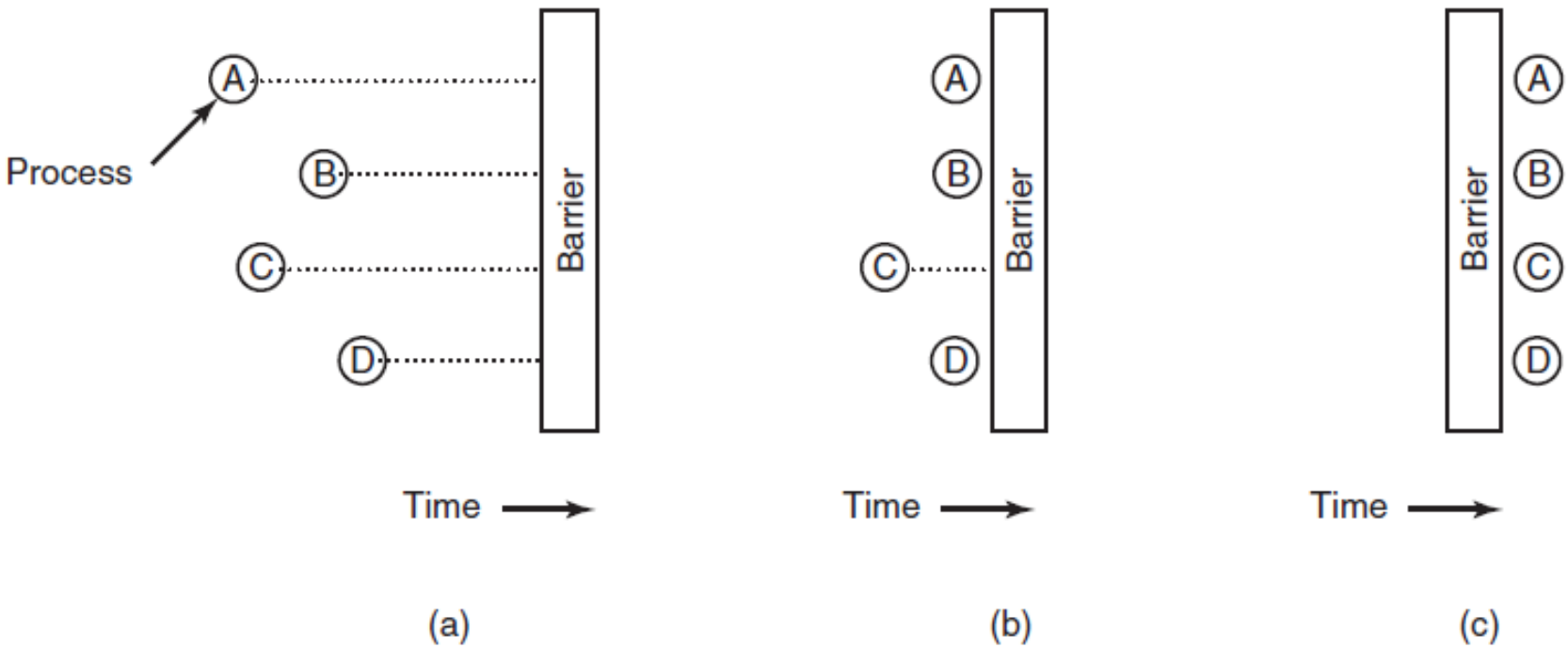
Other Synch. Primitives

- Futex (fast user space mutex)
  - A futex is a feature of Linux that implements basic locking (much like a mutex) but avoids dropping into the kernel unless it really has to.
- Monitors (a higher-level synchronization primitive)

<u>Remark:</u> semaphores are too low level and monitors are not usable except in a few programming languages. Also both do not allow information exchange between machines.

- Message passing primitives such as
  - send(destination, &message);
  - receive(source, &message)
- Barriers

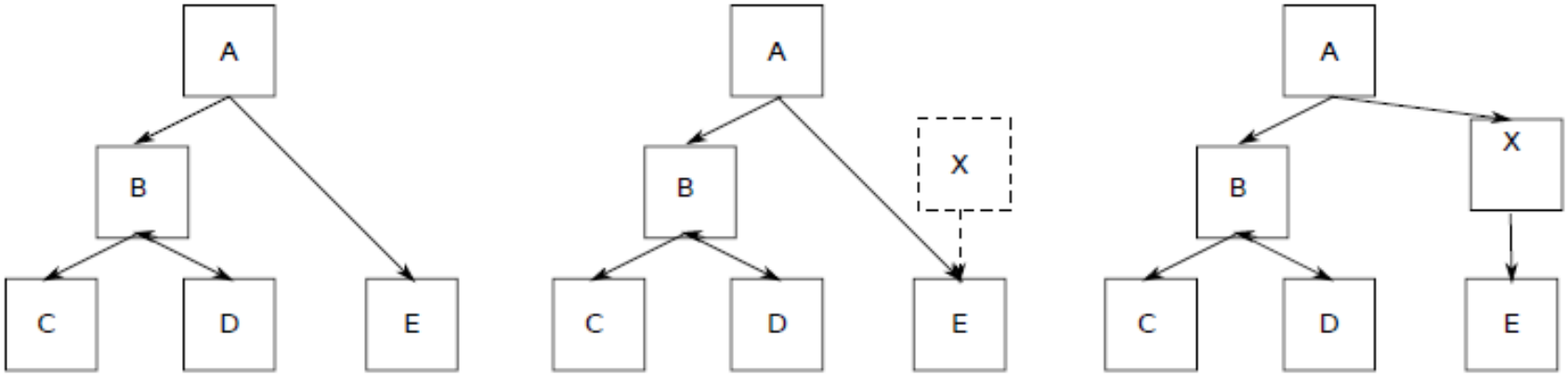
Barriers



Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

Avoiding Locks: Read-Copy-Update (1)

Adding a node:



(a) Original tree

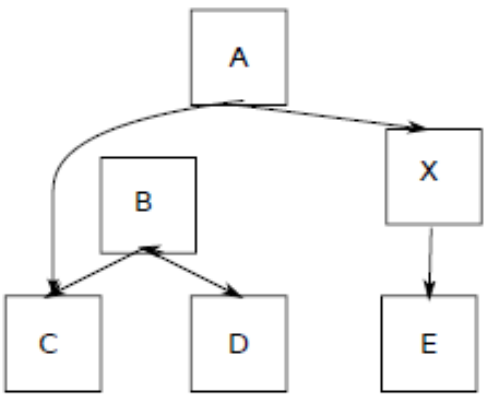
(b) Initialize node X and connect E to X. Any readers in A and E are not affected.

(c) When X is completely initialized, connect X to A. Readers currently in E will have read the old version, while readers in A will pick up the new version of the tree.

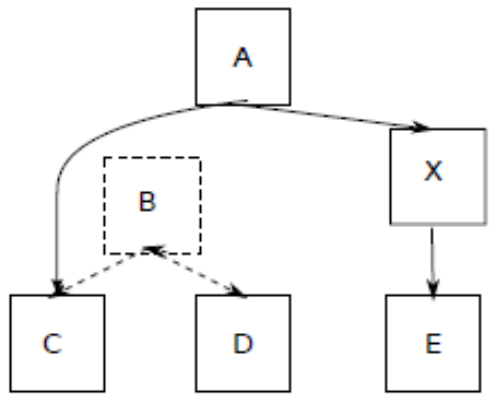
Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

Avoiding Locks: Read-Copy-Update (2)

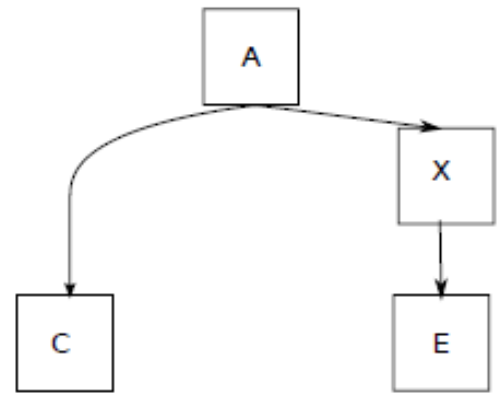
Removing nodes:



(d) Decouple B from A. Note that there may still be readers in B. All readers in B will see the old version of the tree, while all readers currently in A will see the new version.



(e) Wait until we are sure that all readers have left B and C. These nodes cannot be accessed by anymore.

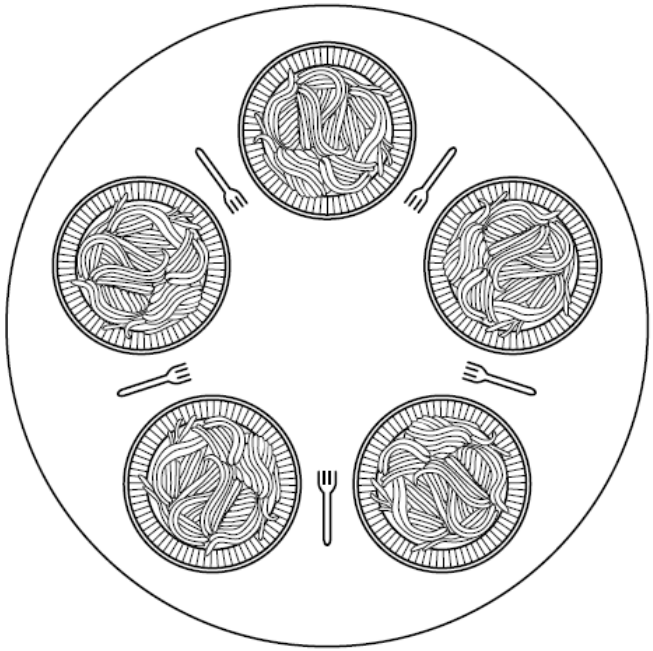


(f) Now we can safely remove B and D

Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

Dining Philosophers Problem (1)

- *n* philosophers spend their lives thinking and eating in a room.
- In their round table, each have their plates full of spagetti and *n* forks.
- They need two forks two eat.



Lunch time in the Philosophy Department.

Dining Philosophers Problem (2)

- From time to time, a philosopher gets hungry.
- He tries to pick up the two forks that are on his right and his left.
- A philosopher may pick one fork at a time.
- A philosopher that picks both forks successfully starts eating.
- When a philosopher finishes eating, he puts down both of his forks to their original position and starts thinking again.

- **Write a program which does not let any philosopher to die due to hunger (i.e. no deadlocks).**

Dining Philosophers Problem (3)

```
#define N 5                                  /* number of philosophers */

void philosopher(int i)                      /* i: philosopher number, from 0 to 4 */
{
        while (TRUE) {
                think( );                    /* philosopher is thinking */
                take_fork(i);                /* take left fork */
                take_fork((i+1) % N);        /* take right fork; % is modulo operator */
                eat( );                      /* yum-yum, spaghetti */
                put_fork(i);                 /* put left fork back on the table */
                put_fork((i+1) % N);         /* put right fork back on the table */
        }
}
```

A nonsolution to the dining philosophers problem.

- Unfortunately, this program fails (deadlock) when all philosophers take their left forks, simultaneously.
- None of them will be able to get their right forks and they will starve.

Dining Philosophers Problem (4)

```
#define  N  5                          /* number of philosophers */
semaphore fork[N]                      /* a semaphore for each chopstick,
                                                each to be initialized to 1 */
```

```
void philosopher(int i)                /* which philosopher (0 to N-1) ? */
{while (TRUE)
 { think();                            /* philosopher is thinking (non-CS)*/
   down(fork[i]);                      /* take left chopstick */
   down(fork[(i+1) % N]);              /* take right chopstick */
   eat();                              /* yum-yum, rice   (CS) */
   up(fork[i]);                        /* put left chopstick */
   up(fork[(i+1) % N]);                /* put left chopstick */
 }
}
```

Still a nonsolution to the dining philosophers problem.

Dining Philosophers Problem (5)

```
#define N            5            /* number of philosophers */
#define LEFT         (i+N–1)%N    /* number of i's left neighbor */
#define RIGHT        (i+1)%N      /* number of i's right neighbor */
#define THINKING     0            /* philosopher is thinking */
#define HUNGRY       1            /* philosopher is trying to get forks */
#define EATING       2            /* philosopher is eating */
typedef int semaphore;            /* semaphores are a special kind of int */
int state[N];                     /* array to keep track of everyone's state */
semaphore mutex = 1;              /* mutual exclusion for critical regions */
semaphore s[N];                   /* one semaphore per philosopher */

void philosopher(int i)           /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                /* repeat forever */
        think( );                 /* philosopher is thinking */
        take_forks(i);            /* acquire two forks or block */
        eat( );                   /* yum-yum, spaghetti */
        put_forks(i);             /* put both forks back on table */
    }
}
```

A solution to the dining philosophers problem.

- *State* array keeps track of whether a philosopher is eating, thinking, or hungry (trying to acquire chopsticks).
- A philosopher may move into eating state only if neither neighbour is eating. The neighbors are defined by the macros LEFT and RIGHT.

Dining Philosophers Problem (6)

```
            put_forks(i);                    /* put both forks back on table */
        }
    }

    void take_forks(int i)                   /* i: philosopher number, from 0 to N–1 */
    {
        down(&mutex);                        /* enter critical region */
        state[i] = HUNGRY;                   /* record fact that philosopher i is hungry */
        test(i);                             /* try to acquire 2 forks */
        up(&mutex);                          /* exit critical region */
        down(&s[i]);                         /* block if forks were not acquired */
    }

    void put_forks(i)                        /* i: philosopher number, from 0 to N–1 */
```

A solution to the dining philosophers problem.

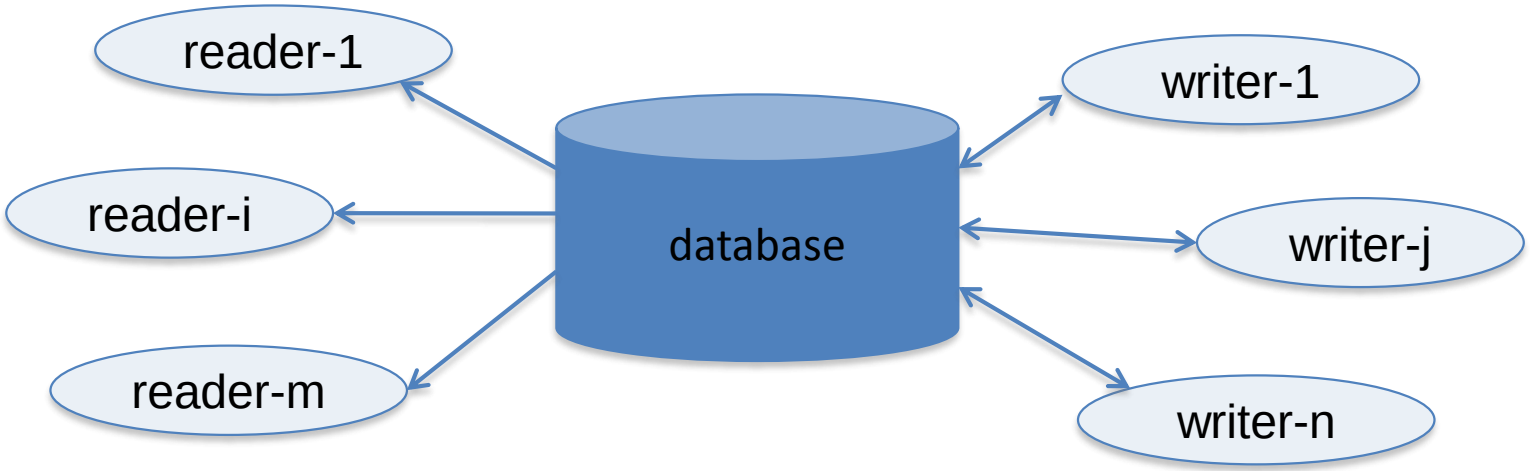
Dining Philosophers Problem (7)

```
}

void put_forks(i)                          /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                      /* enter critical region */
        state[i] = THINKING;               /* philosopher has finished eating */
        test(LEFT);                        /* see if left neighbor can now eat */
        test(RIGHT);                       /* see if right neighbor can now eat */
        up(&mutex);                        /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

A solution to the dining philosophers problem.

Readers and Writers Problem (1)

- Imagine a big database, such as an airline reservation system, with many competing processes wishing to read and write.
- Multiple processes can read database at the same time.
- If one process is writing to the database, no other processes may have access to the database, not even readers.



Readers and Writers Problem (2)

```
typedef int semaphore;
semaphore mutex = 1;     /* controls access to rc */
semaphore db = 1 ;       /* controls access to db */
          int rc = 0 ;   /* no. of processes reading */
```

```
void writer(int j)
{
  think_up_data();       /* non-CS */
  down (db);             /* get exclusive access */
  write_database();      /* update the database */
  up(db);                /* release exclusive access */

}
```

Readers and Writers Problem (3)

```
void reader(int i)
{


                down(db);

  read_database();        /* access the data */


                up(db);

  use_data_read();        /* non-CS */
          }
```

Readers and Writers Problem (4)

```
void reader(int i)
{

                down(db);

  read_database();        /* access the data */

                up(db);

  use_data_read();        /* non-CS */
          }
```

Blocks also other readers!!

Readers and Writers Problem (5)

```
void reader(int i)
{

            rc = rc + 1;        /* one reader more now */
  if (rc == 1) down(db);        /* whether this is the first reader */

  read_database();              /* access the data */

  rc = rc – 1;                  /* one reader fewer now */
  if (rc == 0) up(db);          /* whether this is the last reader */

  use_data_read();              /* non-CS */
        }
```

Readers and Writers Problem (6)

```
void reader(int i)
{

        rc = rc + 1;        /* one reader more now */
  if (rc == 1) down(db);    /* whether this is the first reader */

  read_database()           /* access the data */

  rc = rc – 1;              /* one reader fewer now */
  if (rc == 0) up(db);      /* whether this is the last reader */

  use_data_read()           /* non-C
        }
```

Read count rc is common to readers so reading and updating rc should be protected in CS

Readers and Writers Problem (7)

```
void reader(int i)
{
   down(mutex);              /* get exclusive access to rc */
              rc = rc + 1;   /* one reader more now */
   if (rc == 1) down(db);    /* whether this is the first reader */
   up(mutex);                /* release exclusive access to rc */
   read_database();          /* access the data */
   down(mutex);              /* get exclusive access to rc */
   rc = rc – 1;              /* one reader fewer now */
   if (rc == 0) up(db);      /* whether this is the last reader */
   up (mutex);               /* release exclusive access to rc */
   use_data_read();          /* non-CS */
            }
```

Sleeping Barber Problem

- The barber shop has one barber, one barber chair, and N chairs for waiting customers, if any, to sit in.
- If there is no customer at present, the barber sits down in the barber chair and falls asleep.
- When a customer arrives, he has to wake up the sleeping barber.
- If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there is an empty chair) or leave the shop (if all chairs are full).

- The problem is to program the barber and the customers without getting into race conditions or deadlocks.

Sleeping Barber Problem

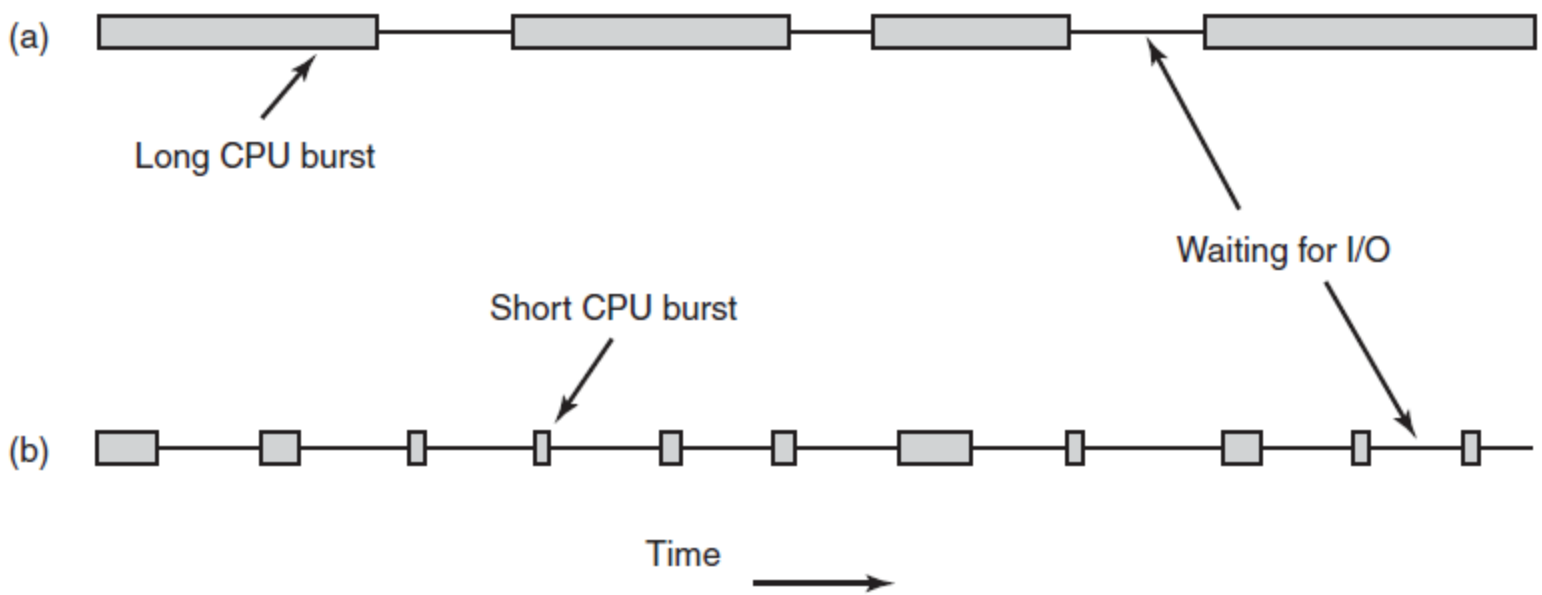
```
#define CHAIRS 5                    /* no. of chairs for waiting customers */
typedef int semaphore;
semaphore customers = 0;            /* no.  of  customers */
semaphore barbers = 0;              /* no. of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion for “waiting” */
int waiting = 0;                    /* customers are waiting not being haircut */
```

```
void Barber(void)
{while (TRUE)
  {down(customers);                 /* go to sleep if number of customers is 0 */
   down(mutex);                     /* acquire access to ‘waiting’ */
   waiting = waiting – 1;           /* decrement count of waiting customers */
   up(barbers);                     /* one barber is now ready to cut hair */
   up(mutex);                       /* release ‘waiting’ */
   cut_hair();                      /* cut hair, non-CS */
  }
}
```

Sleeping Barber Problem

```
void customer(void)
{
down(mutex);                    /* enter CS */
 if (waiting < CHAIRS)          /* increment count of waiting customers */
{
   waiting = waiting + 1;
   up(customers);               /* wake up barber if necessary */
   up(mutex);                   /* release access to 'waiting' */
   down(barbers);               /* wait if no free barbers */
   get_haircut();               /* non-CS */
            }
else
{
up(mutex);                      /* shop is full, do not wait, exit shop */
}
}
```

Introduction to Scheduling
Process Behavior



Bursts of CPU usage alternate with periods of waiting for I/O.
(a) A CPU-bound process. (b) An I/O-bound process.

Scheduler

- If we consider batch systems, there is often more processes submitted than the number of processes that can be executed immediately.

- Hence incoming processes are spooled (to a disk).

- **The long-term scheduler** selects processes from this process pool and loads selected processes into memory for execution.

Scheduler

- **Short-term scheduler** selects the process to get the CPU from among the processes which are already in memory.

- Short-time scheduler executes frequently (mostly at least once every 10 milliseconds).
  - It executes on CPU and has to be very fast.

  - If it takes 1 millisecond to choose a process, this means (1 / (10 + 1)) = 9% of the processor time is being used for short term scheduling and only 91% may be used by user processes for execution.

Scheduler

- Long-term scheduler on the other hand executes much less frequently.
- It controls the degree of multiprogramming (no. of processes in memory at a time).
- If the degree of multiprogramming is to be kept stable (say 10 processes at a time), then the long-term scheduler may only need to be invoked when a process finishes execution.

Scheduler

- Time-sharing systems (mostly) have no long-term scheduler.
- The stability of these systems either depends upon a physical limitation (no. of available terminals) or the self-adjusting nature of users (if you can't get response, you quit).
- It can sometimes be good to reduce the degree of multiprogramming by removing processes from memory and storing them on disk.
- These processes can then be reintroduced into memory by the **medium-term scheduler.**
- This operation is also known as **swapping**. Swapping may be necessary to improve the process mix or to free memory.

Categories of Scheduling Algorithms

1. Batch.
2. Interactive.
3. Real time.

Performance Criteria

- In order to achieve an efficient processor management, OS tries to select the most appropriate process from the ready queue.

- For selection, the relative importance of the followings may be considered as performance criteria.

Scheduling Algorithm Goals

All systems

- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

Batch systems

- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

Interactive systems

- Response time - respond to requests quickly
- Proportionality - meet users' expectations

Real-time systems

- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

Some goals of the scheduling algorithm under different circumstances

Performance Criteria

- **Processor Utilization:**

  The ratio of busy time of the processor to total time

$$\textit{Processor Utilization} = \frac{\textit{(Processor busy time}}{\textit{(Processor busy time + Processor idle time)}}$$

- We would like to keep the processor as busy as possible.

Performance Criteria

- **Throughput:**

  The measure of work done in a unit time interval.

  Throughput = (Number of processes completed) / (Time Unit)

Performance Criteria

- **Turnaround Time (tat):**

  The sum of time spent in the ready queue, execution time and I/O time.

  tat = t(process completed) – t(process submitted)

Performance Criteria

- **Waiting Time (wt):**

  Time spent only in ready queue.

  Processor scheduling algorithms only affect the time spent waiting in the ready queue.

  Hence, considering only waiting time instead of turnaround time is generally sufficient.

Performance Criteria

- **Response Time (rt):**

  The amount of time it takes to start responding to a request. This criterion is important for interactive systems.

rt = t(first response) – t(submission of request)

the time that the process starts execution

Performance Criteria

- We, normally, want to
  - maximize
    - processor utilization, throughput
  - minimize
    - tat, wt, rt.
- Sometimes other combinations may also be considered depending on system requirements.
  - For example, for time sharing systems, response time is quite important.

Scheduling in Batch Systems

- First-Come First-Served
- Shortest Job First
- Shortest Remaining Time Next

Processor Scheduling algorithms

- The goal of a scheduling algorithm is to select the most appropriate process in the ready queue.
- For simplicity, we will assume that
  - we have a **single I/O server** and a **single device queue**,
  - device queue always implemented with **FCFS (FIFO)** method.
- We also will **neglect** the **switching time** between processes (*context switching*).

Processor Scheduling algorithms

- We will consider the following processes and will apply different scheduling algorithms on it.

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

First-Come-First-Served (FCFS)

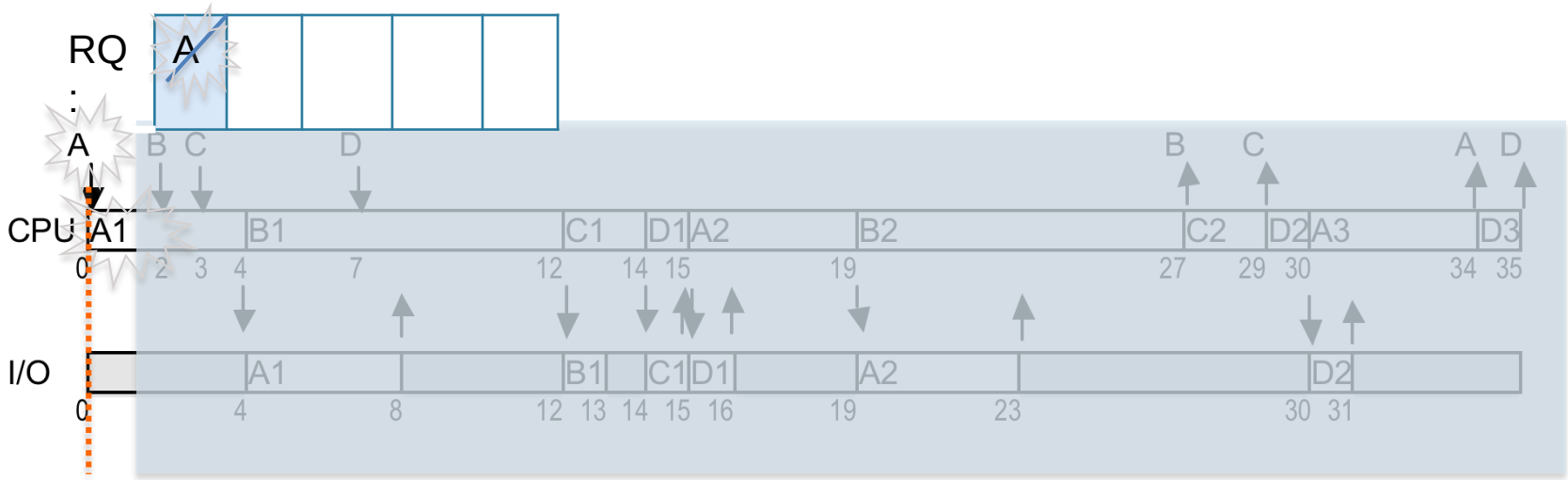
- In this algorithm, the process to be selected is the process which requests the processor first.
- This is the process whose PCB is at the head of the ready queue.
- Contrary to its simplicity, its performance may often be poor compared to other algorithms.

First-Come-First-Served (FCFS)

- FCFS may cause processes with short processor bursts to wait for a long time.
- If one process with a long processor burst gets the processor, all the others will wait for it to release it and the ready queue will be crowded.
- This is called the **convoy** *effect*.

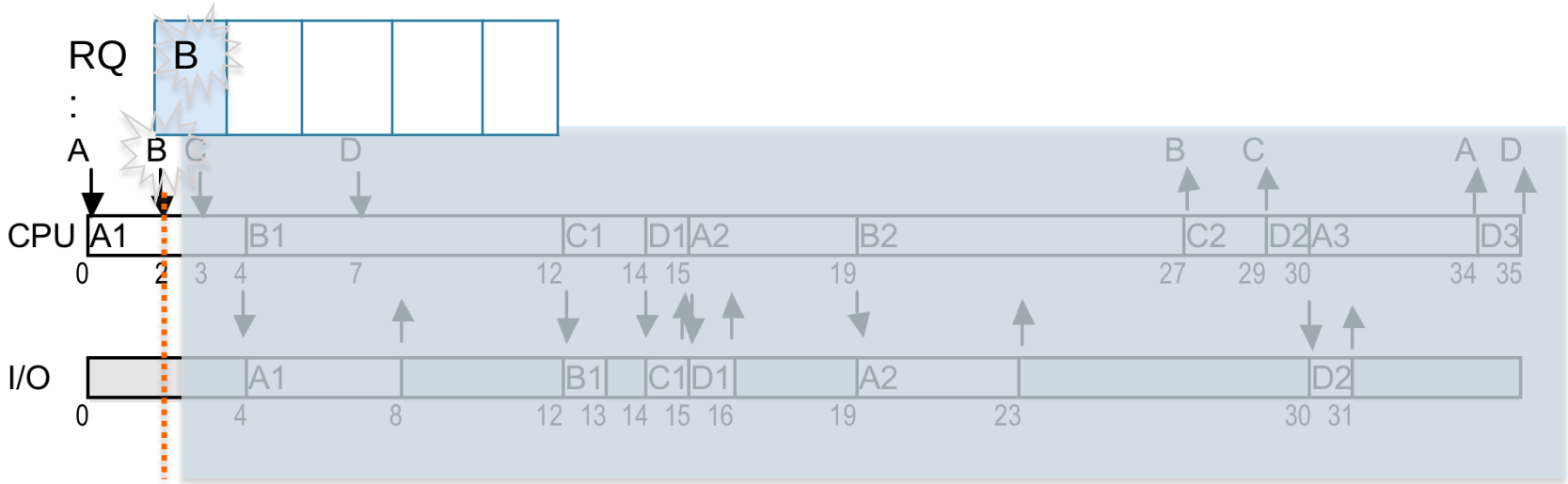
First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



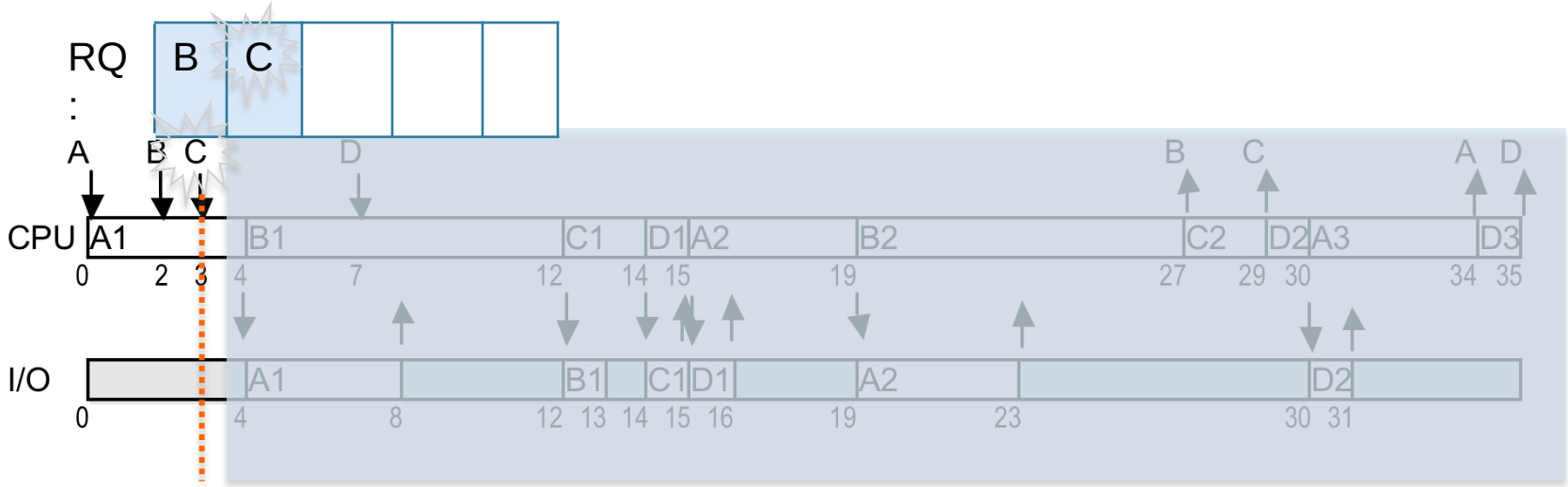
First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



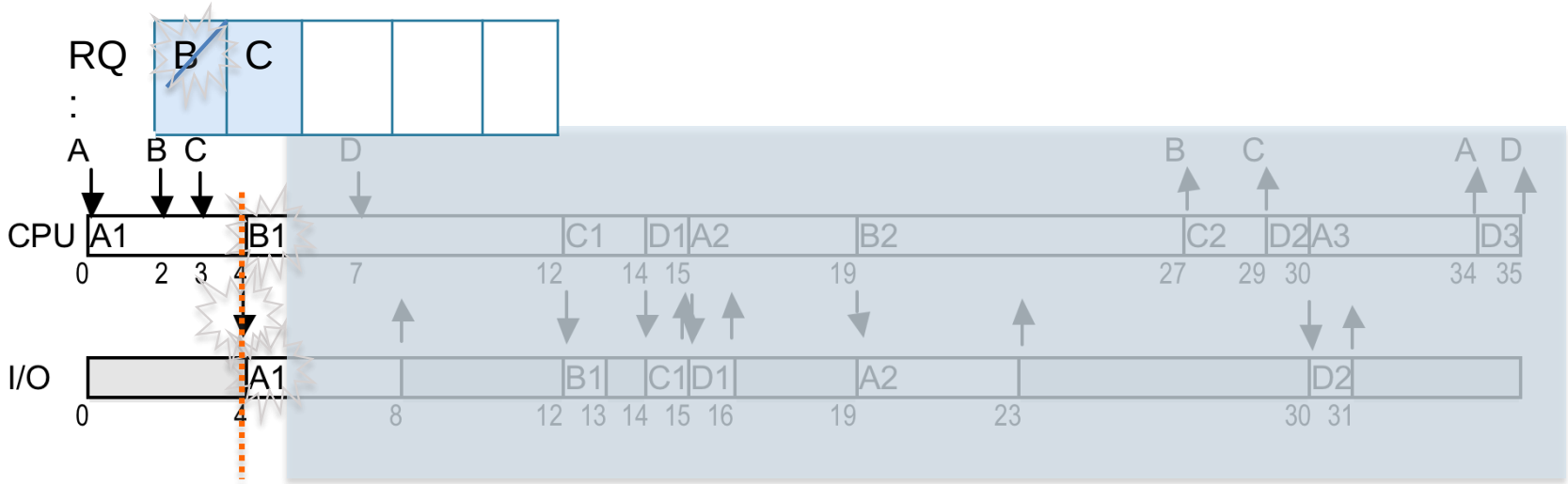
First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



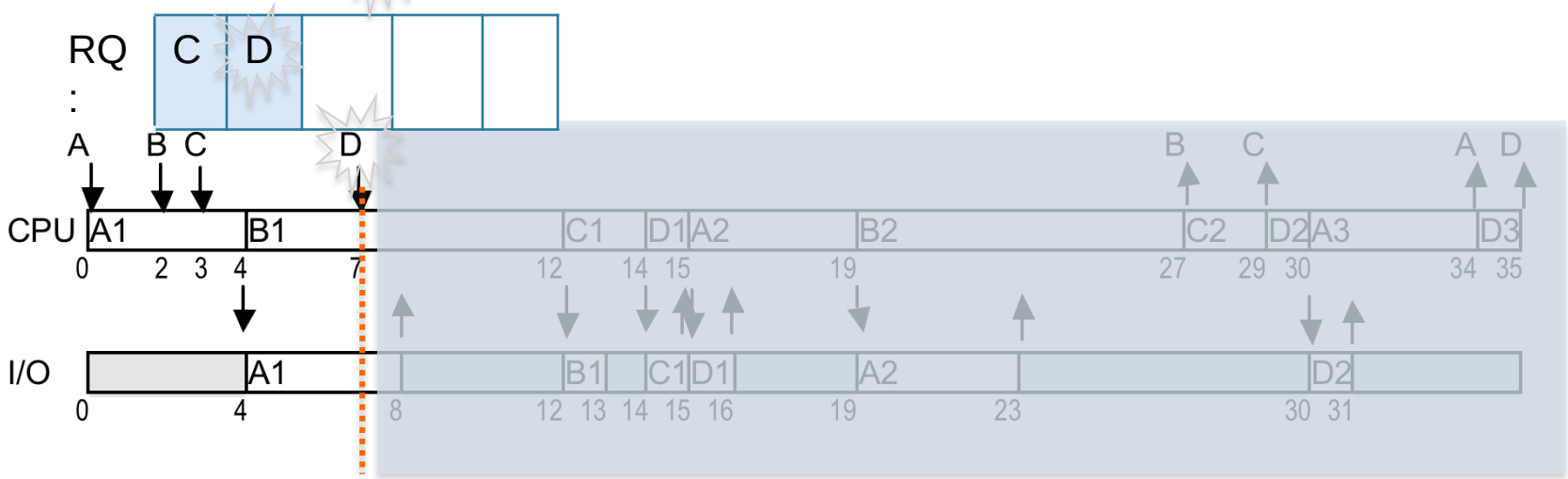
First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



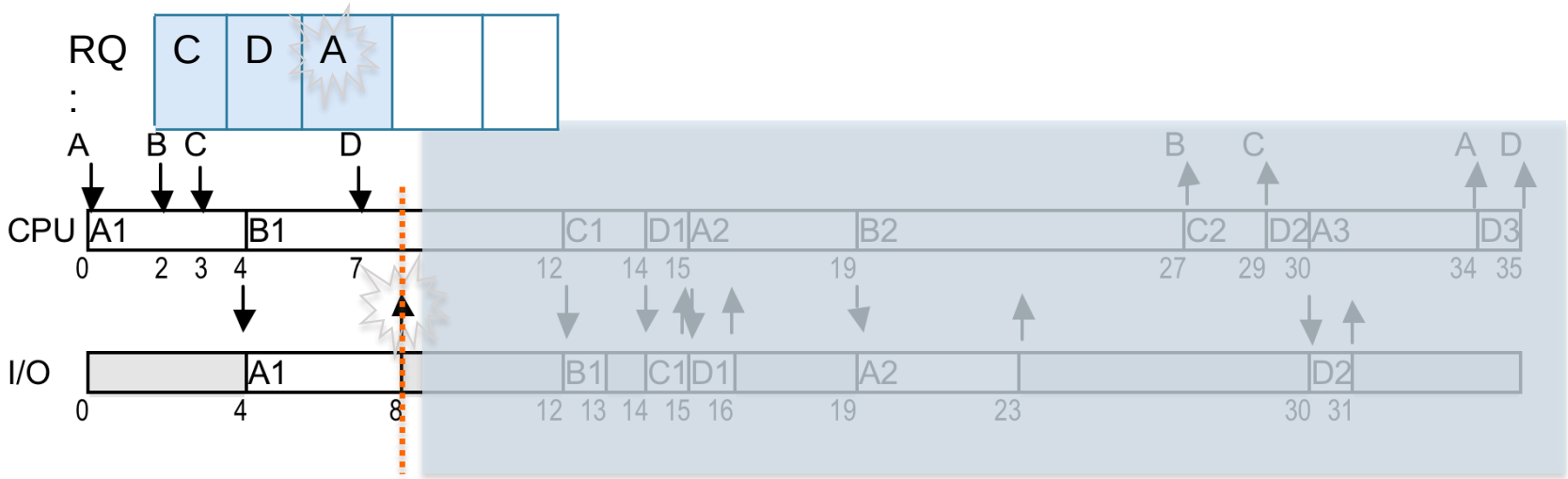
First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



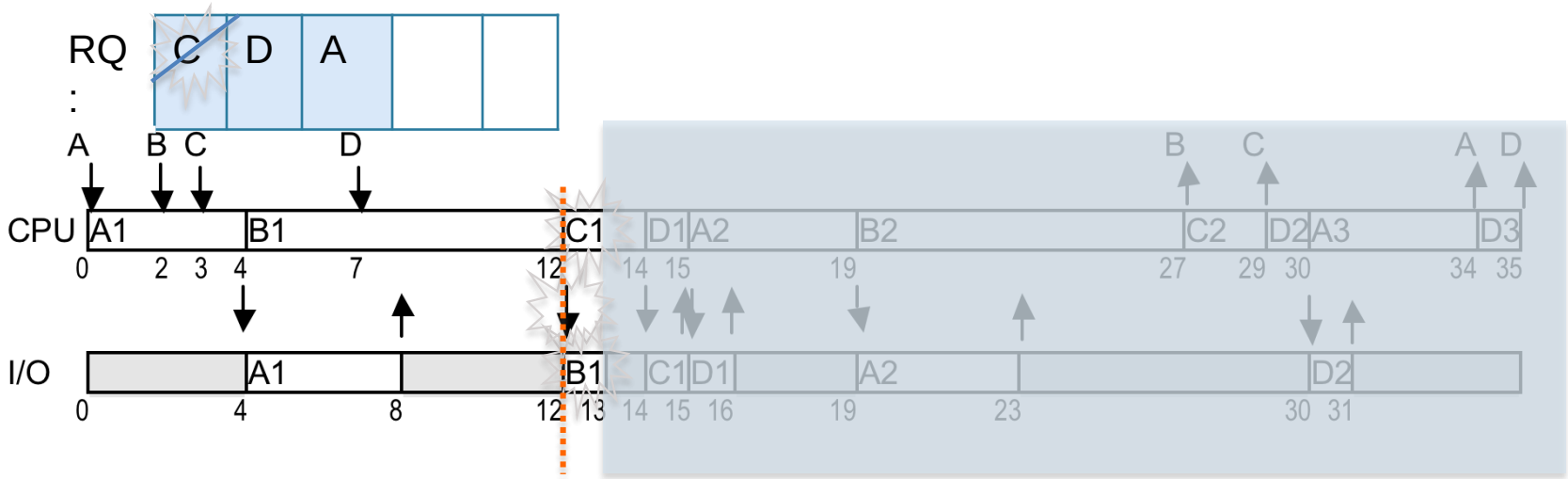
First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



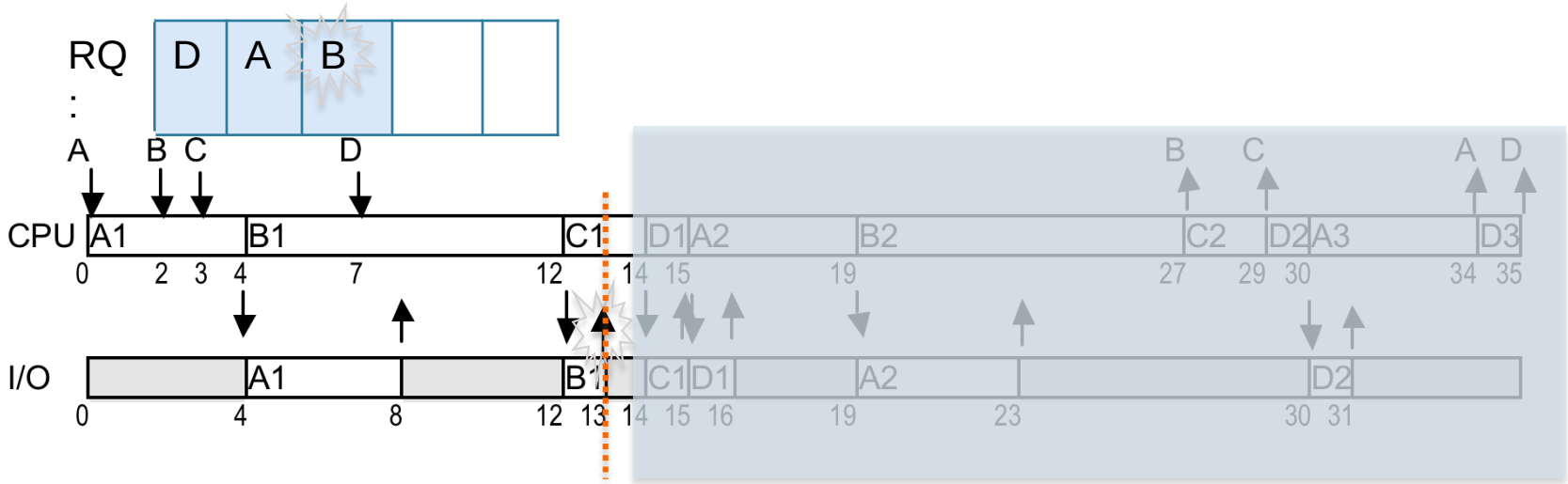
First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



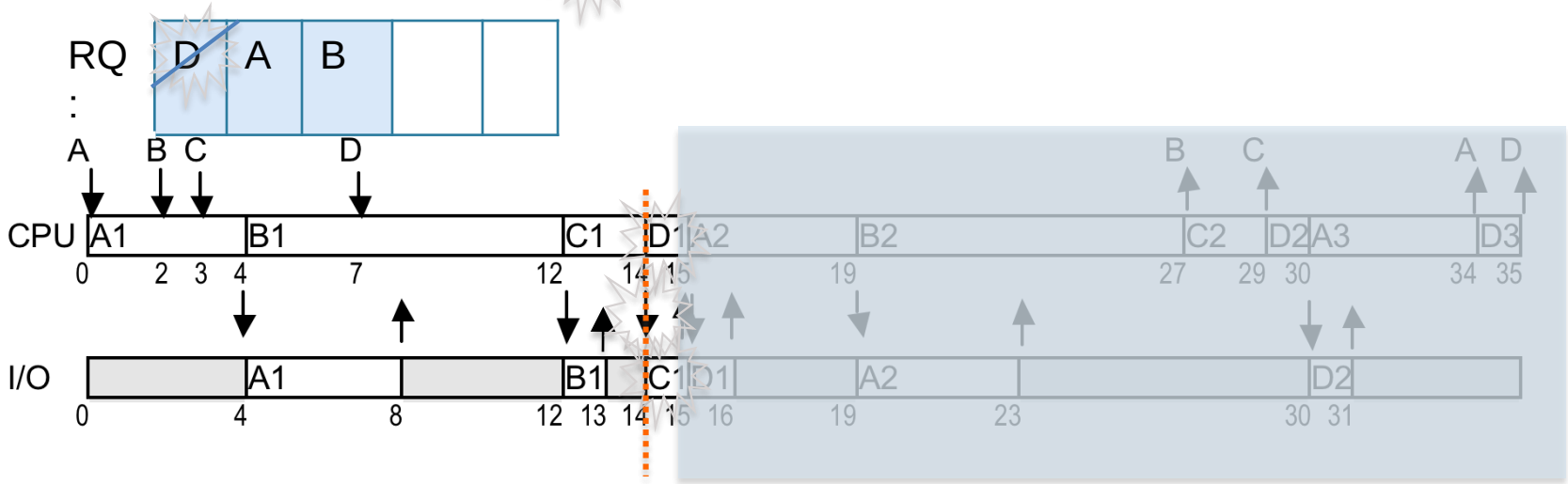
First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



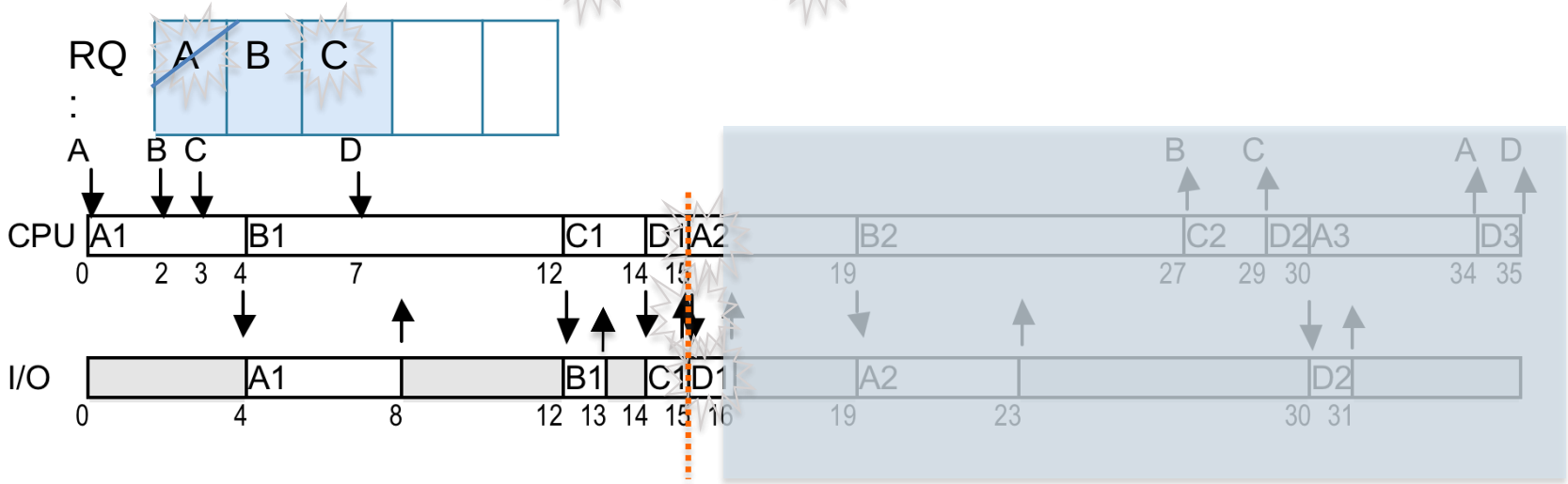
First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



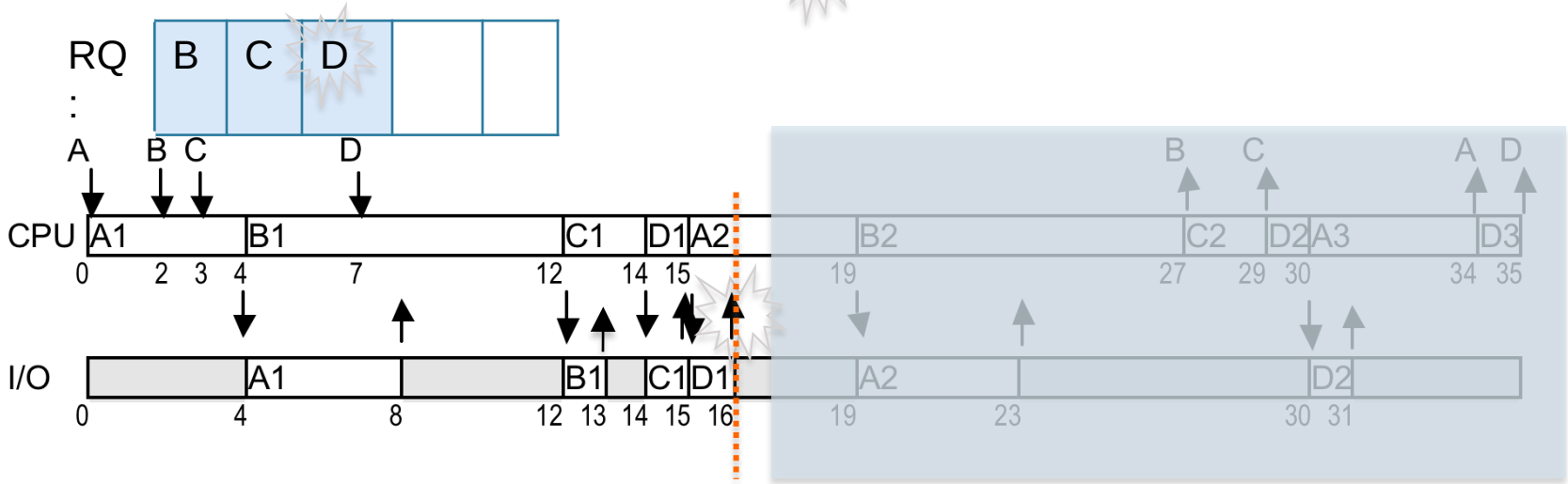
First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



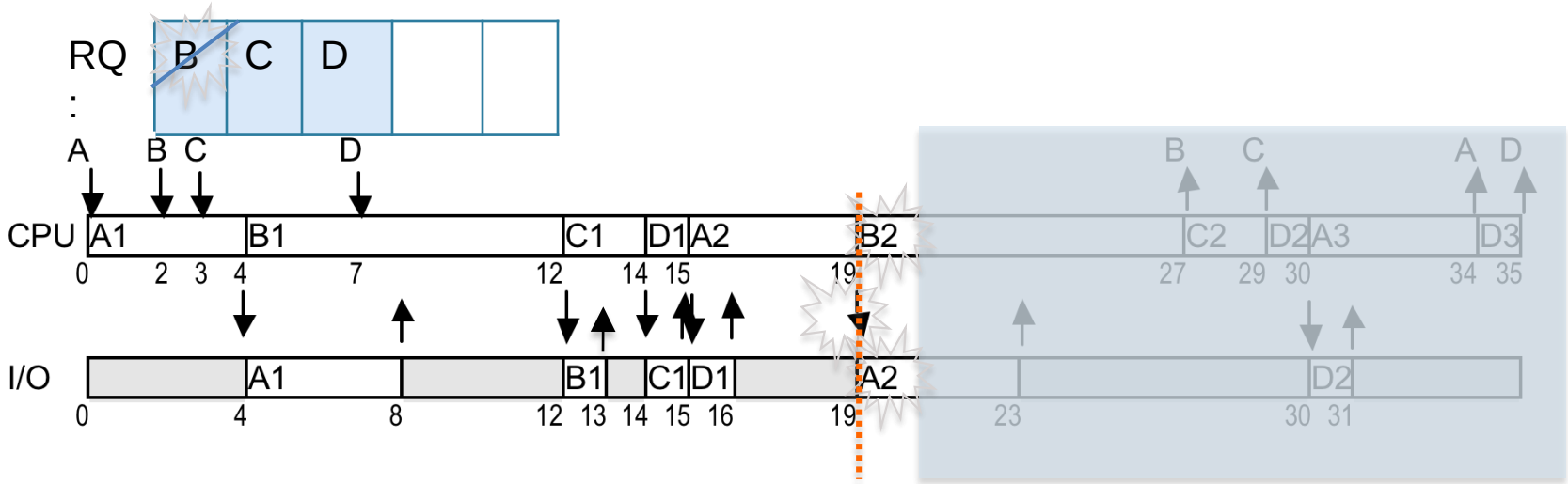
First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



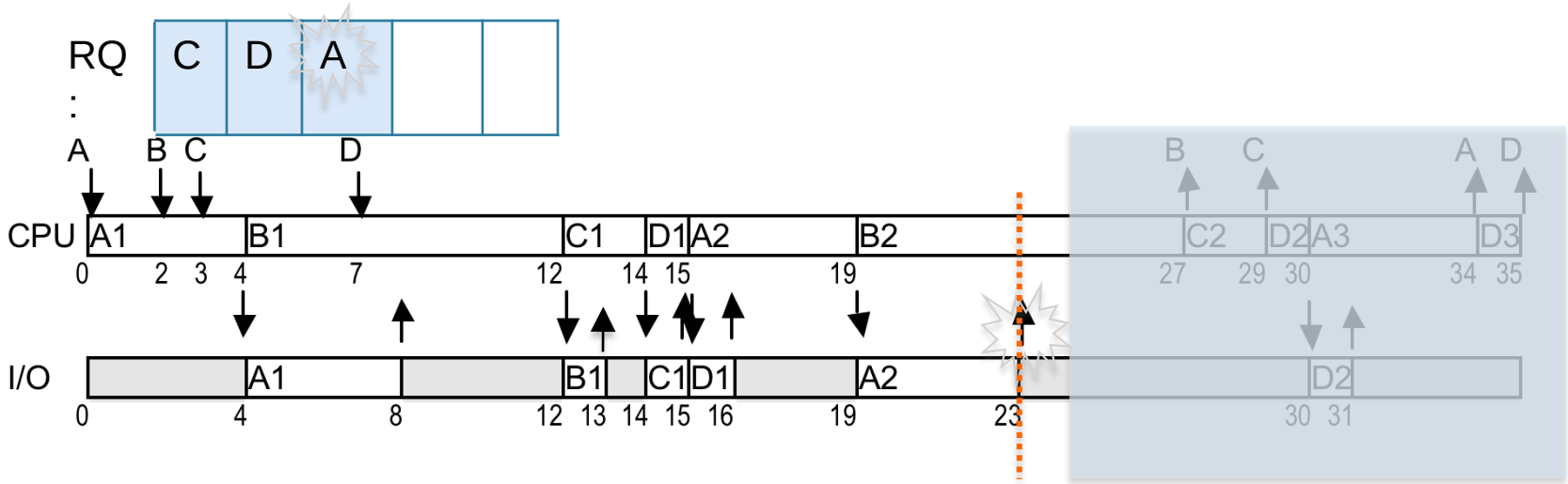
First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	6	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



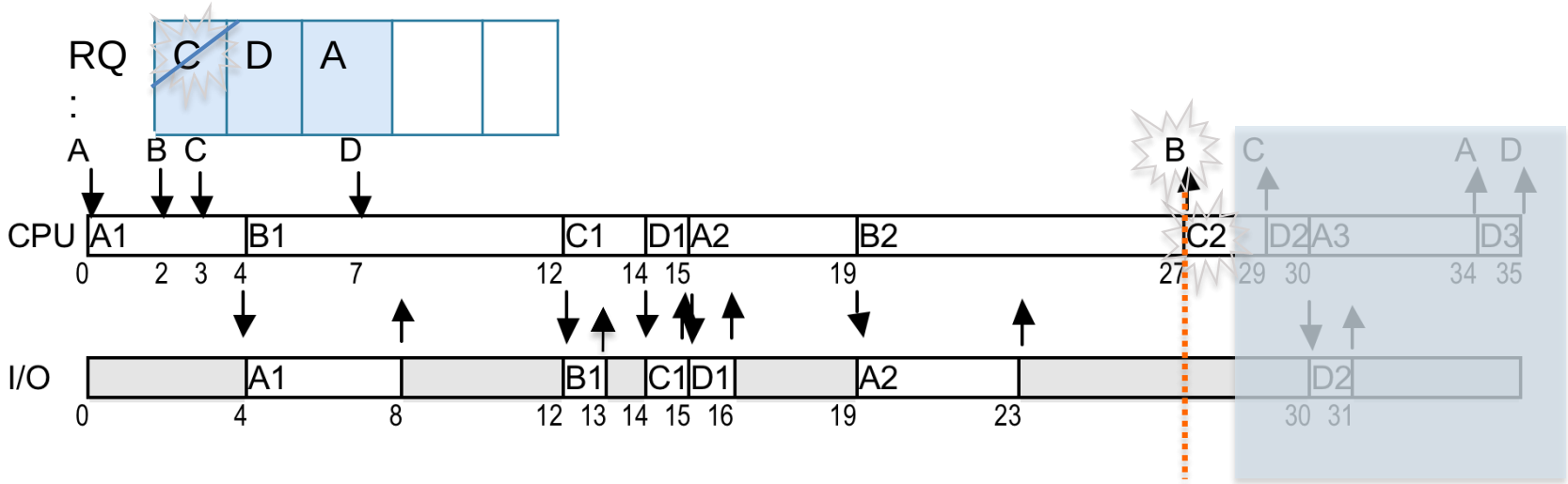
First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



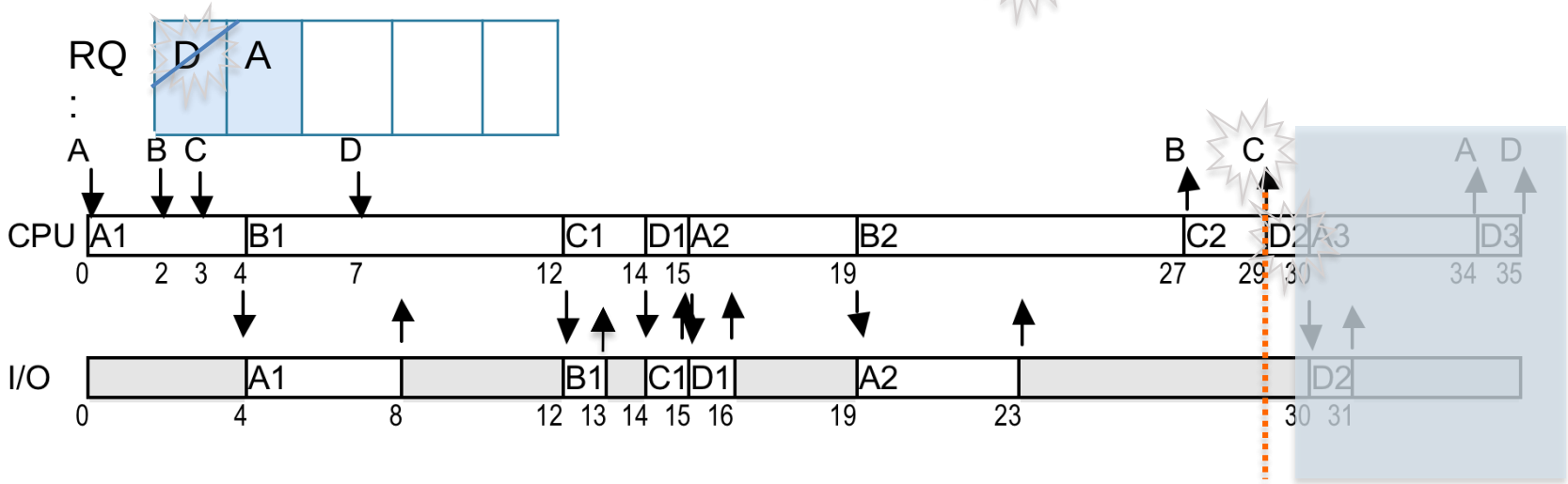
First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



2.3.1 First-Come-First-Served (FCFS)

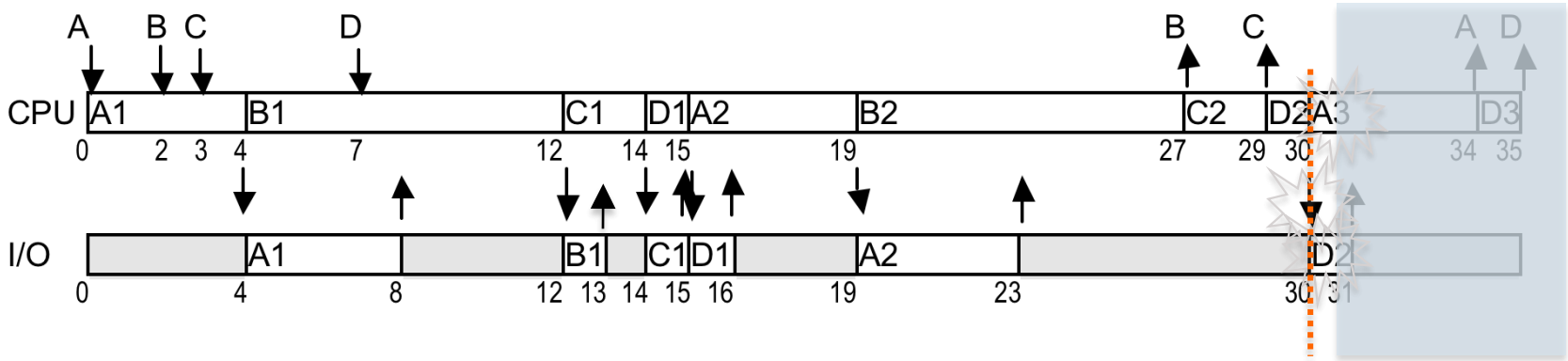
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

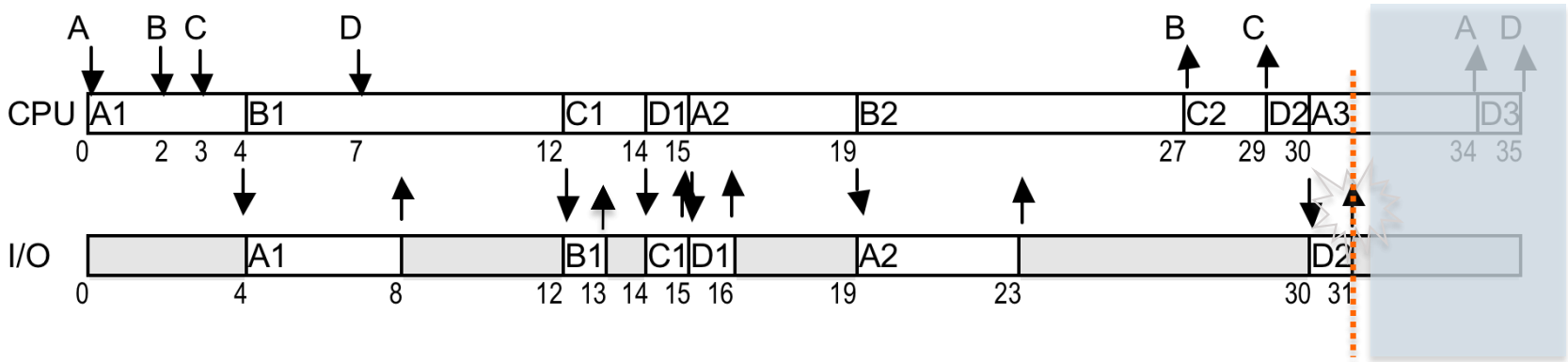
RQ: ~~A~~



First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

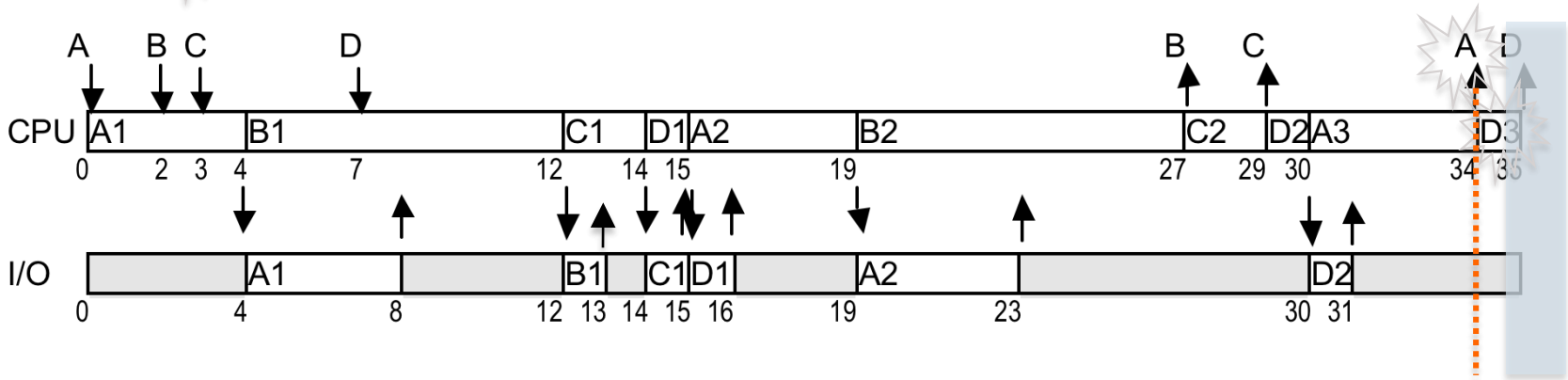
RQ: D



First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

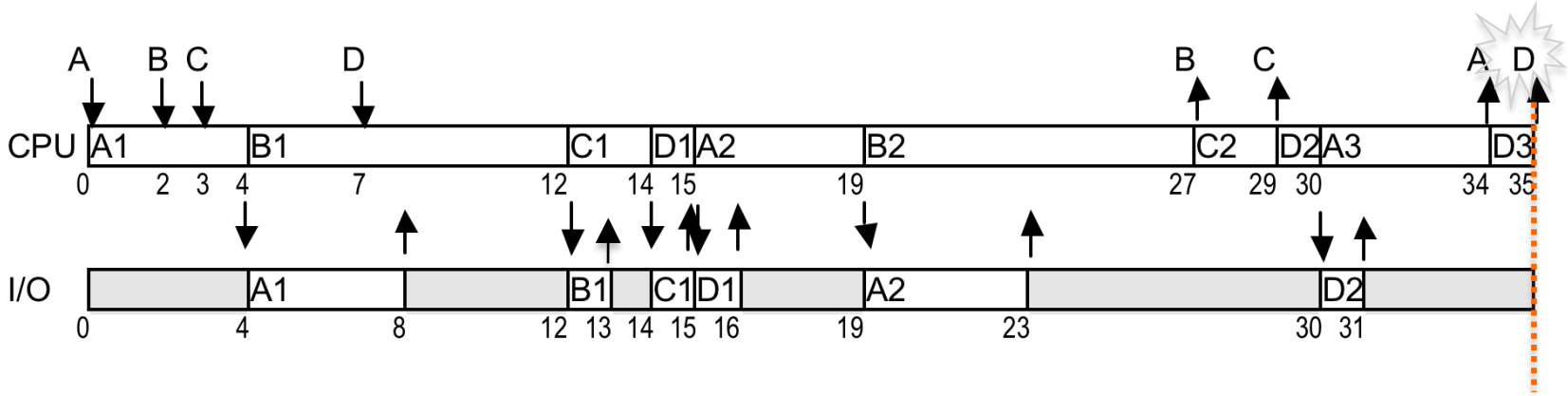
RQ: ~~D~~



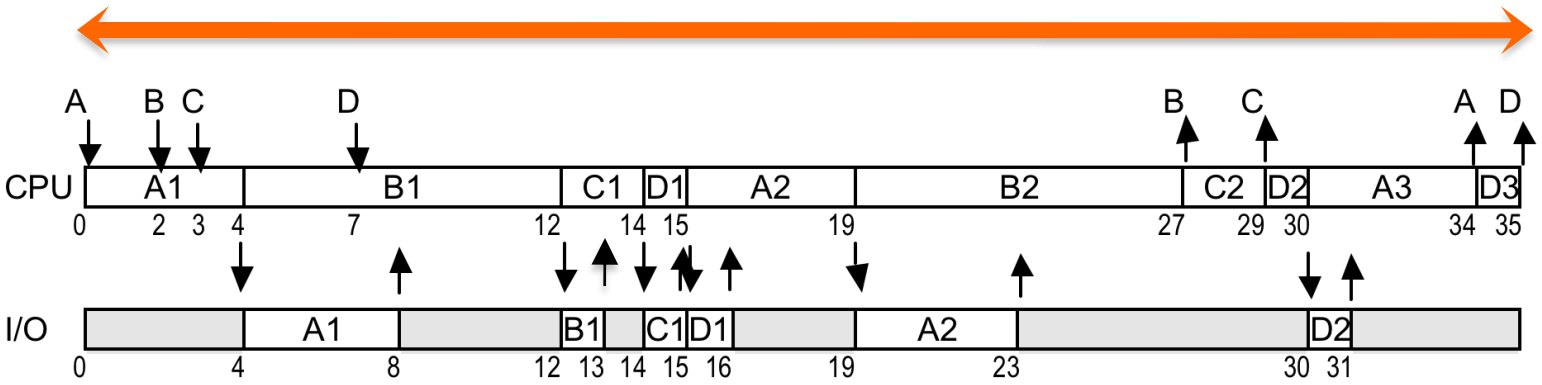
First-Come-First-Served (FCFS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ:

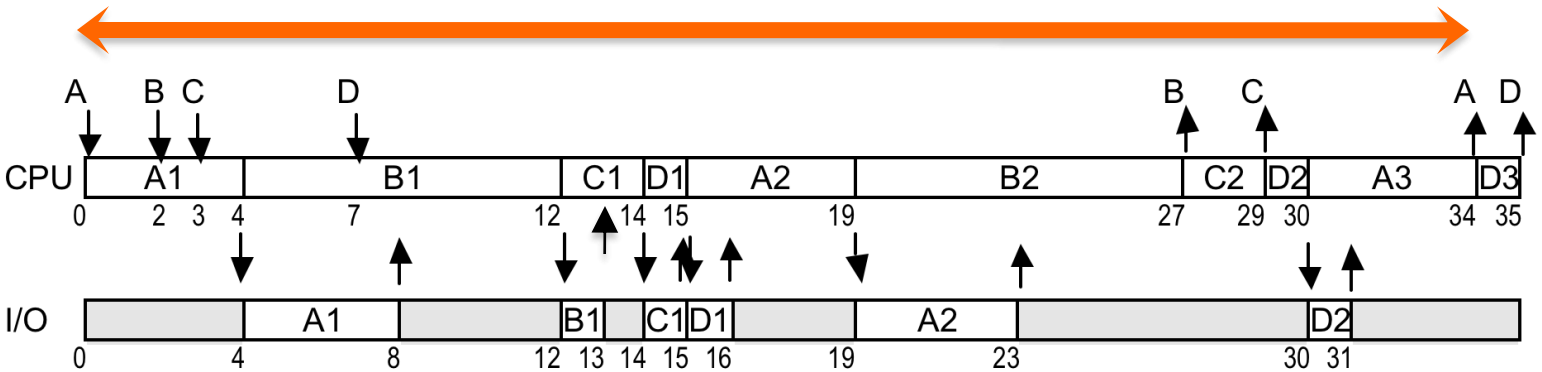


First-Come-First-Served (FCFS)



- Processor utilization = (35 / 35) * 100 = 100 %
- Throughput = 4 / 35=0.11

First-Come-First-Served (FCFS)



- Turn around time:

  $tat_A = 34 - 0 = 34$

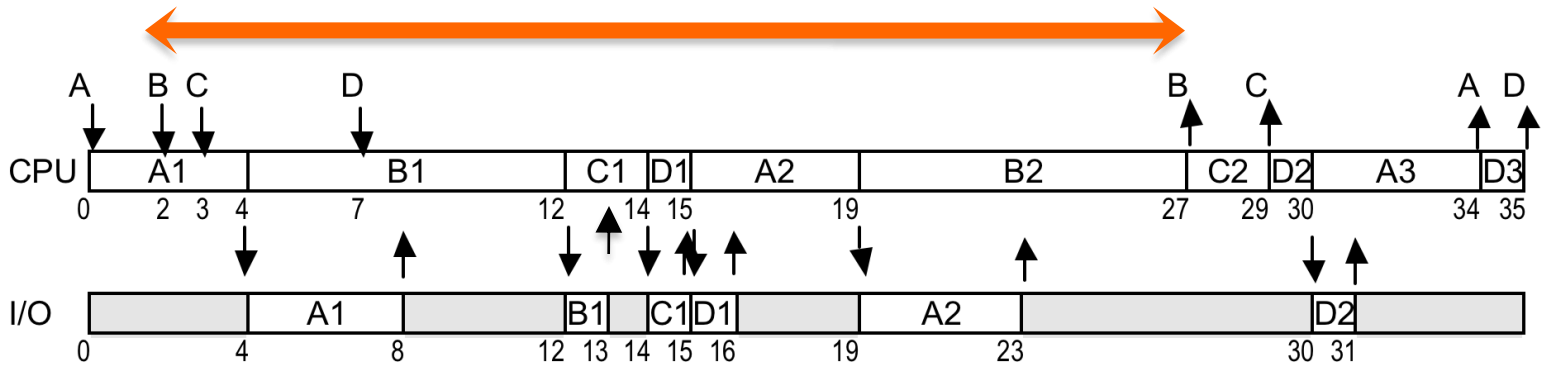
  $tat_B = 27 - 2 = 25$

  $tat_C = 29 - 3 = 26$

  $tat_D = 35 - 7 = 28$

  $tat_{AVG} = (34 + 25 + 26 + 28) / 4 = 28.25$

First-Come-First-Served (FCFS)



- Turn around time:

  $tat_A = 34 - 0 = 34$

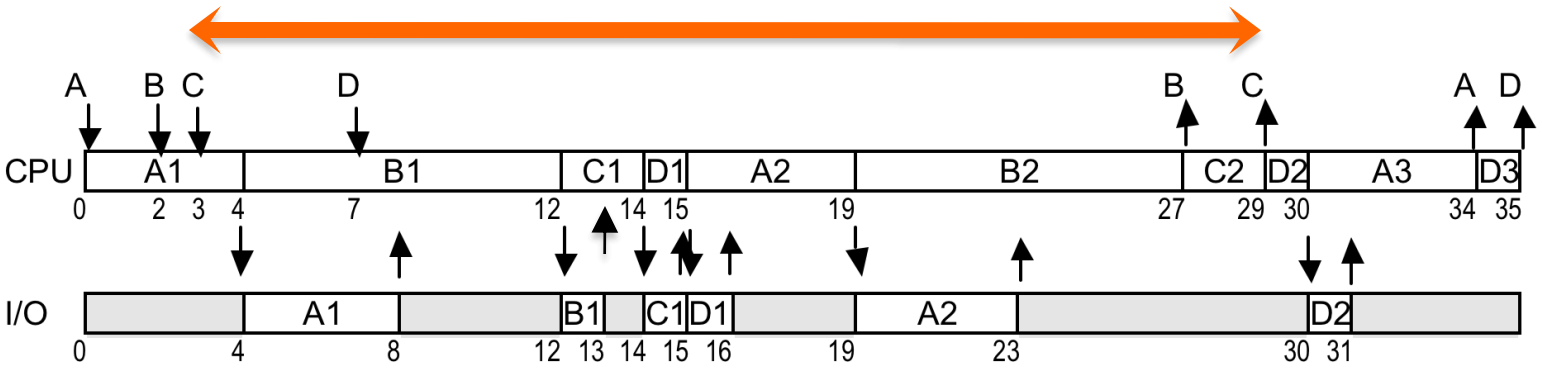
  $tat_B = 27 - 2 = 25$

  $tat_C = 29 - 3 = 26$

  $tat_D = 35 - 7 = 28$

  $tat_{AVG} = (34 + 25 + 26 + 28) / 4 = 28.25$

First-Come-First-Served (FCFS)



- Turn around time:

  $tat_A = 34 - 0 = 34$

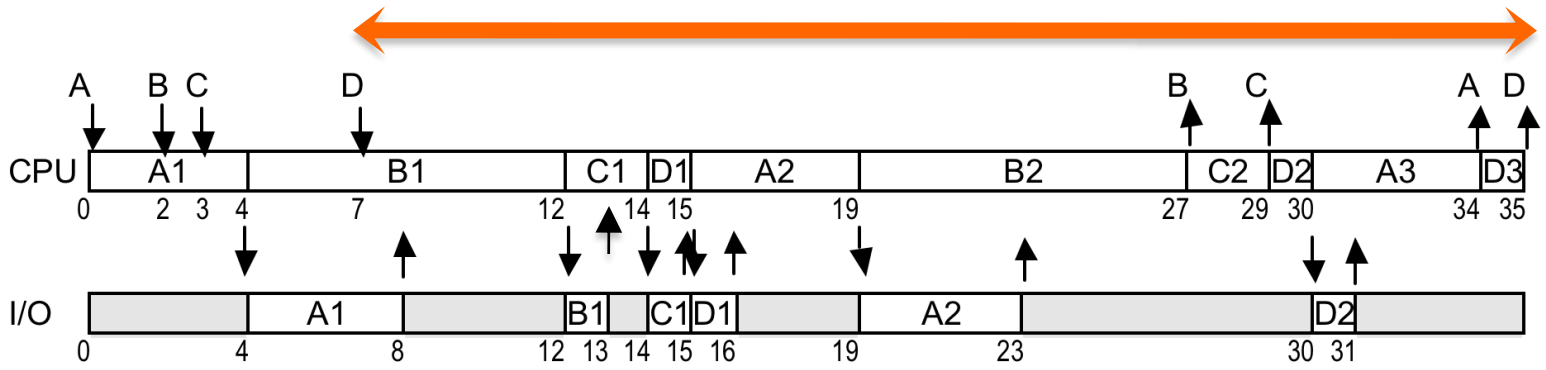
  $tat_B = 27 - 2 = 25$

  $tat_C = 29 - 3 = 26$

  $tat_D = 35 - 7 = 28$

  $tat_{AVG} = (34 + 25 + 26 + 28) / 4 = 28.25$

First-Come-First-Served (FCFS)



- Turn around time:

  $tat_A = 34 - 0 = 34$

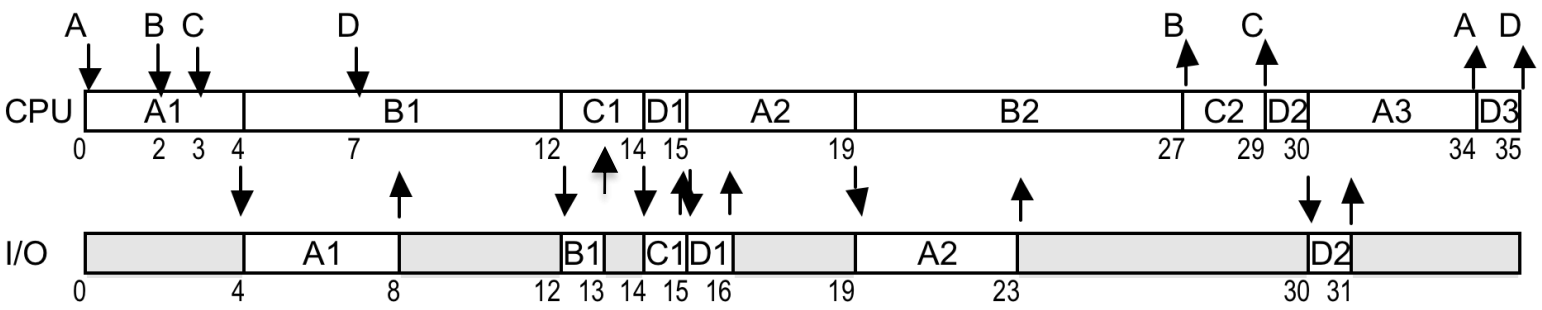
  $tat_B = 27 - 2 = 25$

  $tat_C = 29 - 3 = 26$

  $tat_D = 35 - 7 = 28$

  $tat_{AVG} = (34 + 25 + 26 + 28) / 4 = 28.25$

First-Come-First-Served (FCFS)



- Turn around time:

  $tat_A = 34 - 0 = 34$

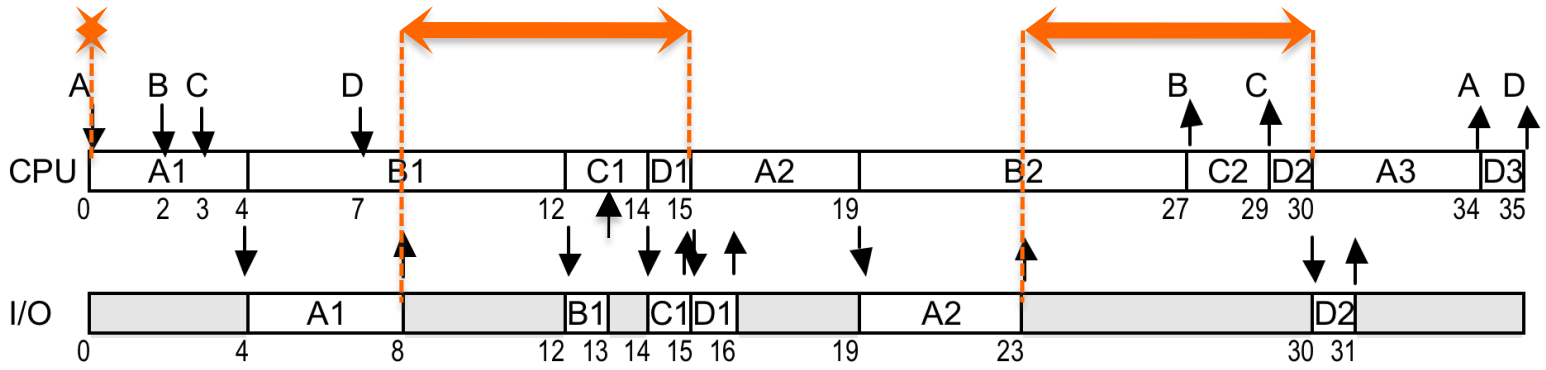
  $tat_B = 27 - 2 = 25$

  $tat_C = 29 - 3 = 26$

  $tat_D = 35 - 7 = 28$

  $tat_{AVG} = (34 + 25 + 26 + 28) / 4 = 28.25$

First-Come-First-Served (FCFS)



- Waiting time:

$wt_A = (0 - 0) + (15 - 8) + (30 - 23) = 14$

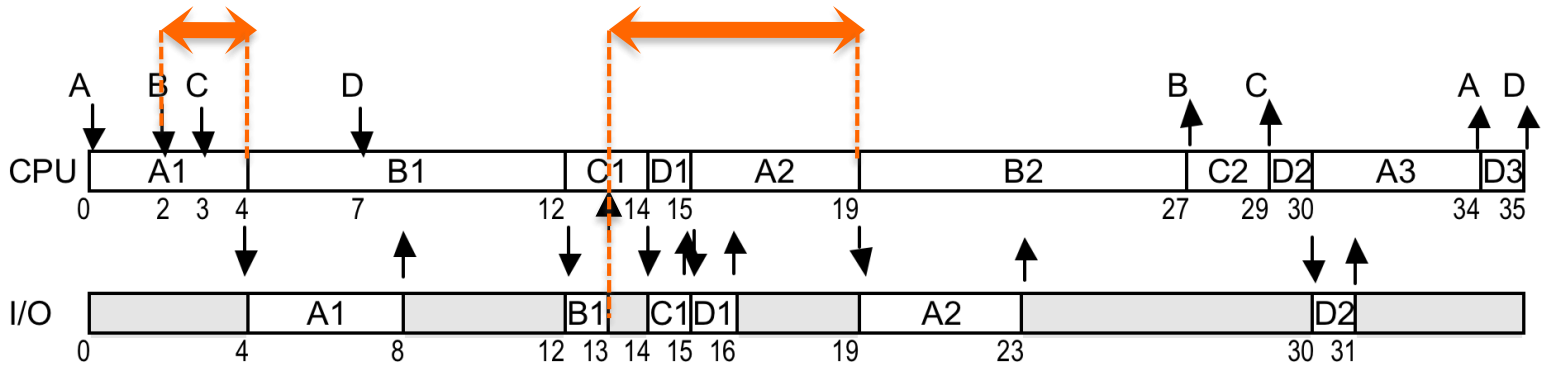
$wt_B = (4 - 2) + (19 - 13) = 8$

$wt_C = (12 - 3) + (27 - 15) = 21$

$wt_D = (14 - 7) + (29 - 16) + (34 - 31) = 23$

$wt_{AVG} = (14 + 12 + 21 + 23) / 4 = 16.5$

First-Come-First-Served (FCFS)



- Waiting time:

$wt_A = (0 - 0) + (15 - 8) + (30 - 23) = 14$

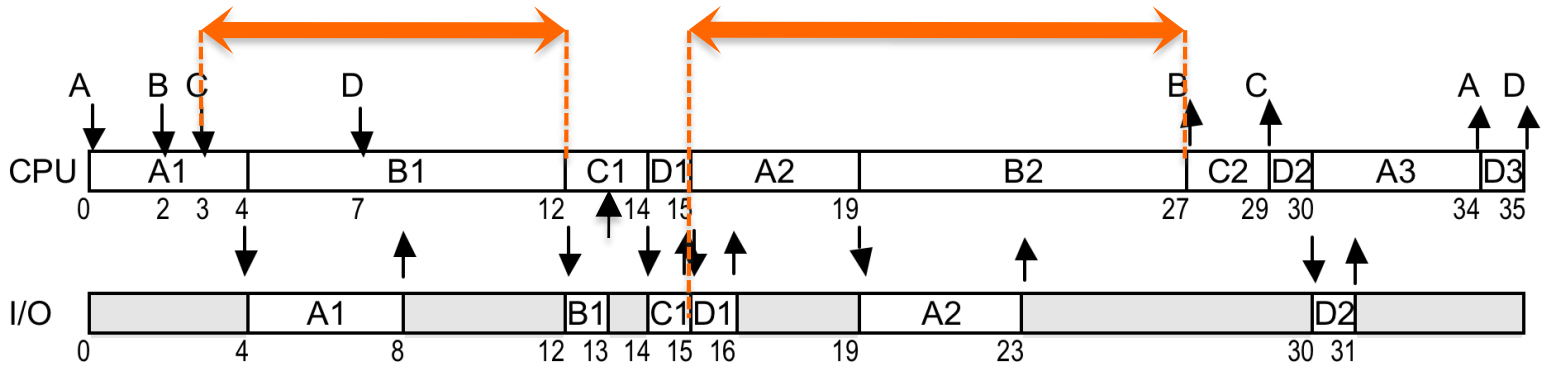
$wt_B = (4 - 2) + (19 - 13) = 8$

$wt_C = (12 - 3) + (27 - 15) = 21$

$wt_D = (14 - 7) + (29 - 16) + (34 - 31) = 23$

$wt_{AVG} = (14 + 12 + 21 + 23) / 4 = 16.5$

First-Come-First-Served (FCFS)



- Waiting time:

$wt_A = (0 - 0) + (15 - 8) + (30 - 23) = 14$

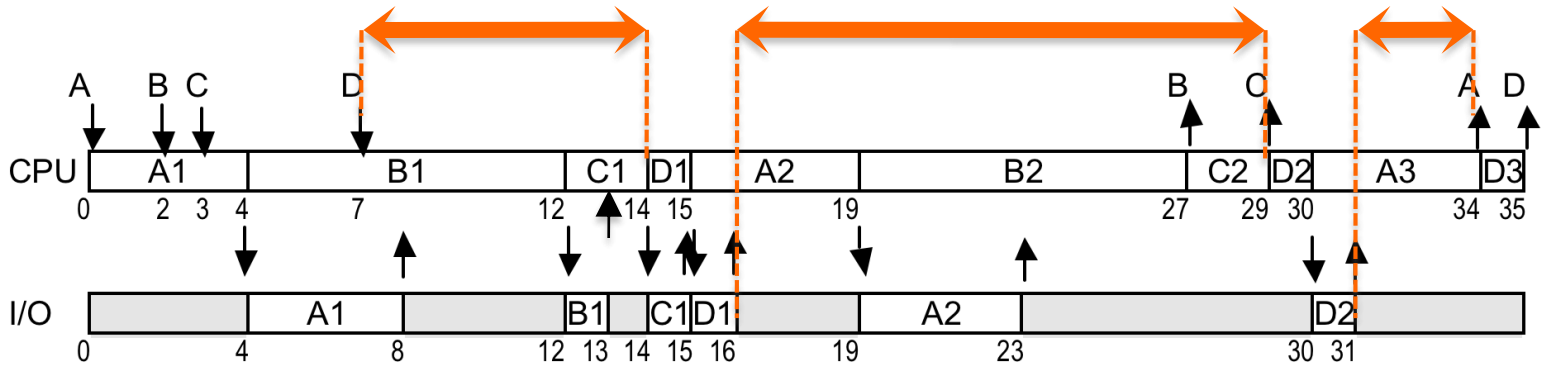
$wt_B = (4 - 2) + (19 - 13) = 8$

$wt_C = (12 - 3) + (27 - 15) = 21$

$wt_D = (14 - 7) + (29 - 16) + (34 - 31) = 23$

$wt_{AVG} = (14 + 12 + 21 + 23) / 4 = 16.5$

First-Come-First-Served (FCFS)



- Waiting time:

$wt_A = (0 - 0) + (15 - 8) + (30 - 23) = 14$

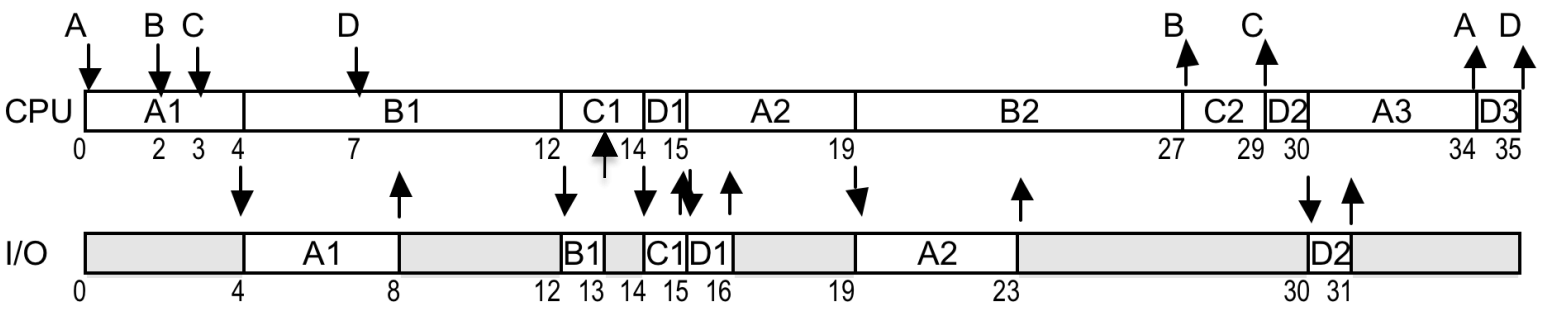
$wt_B = (4 - 2) + (19 - 13) = 8$

$wt_C = (12 - 3) + (27 - 15) = 21$

$wt_D = (14 - 7) + (29 - 16) + (34 - 31) = 23$

$wt_{AVG} = (14 + 12 + 21 + 23) / 4 = 16.5$

First-Come-First-Served (FCFS)



- Waiting time:

$wt_A = (0 - 0) + (15 - 8) + (30 - 23) = 14$

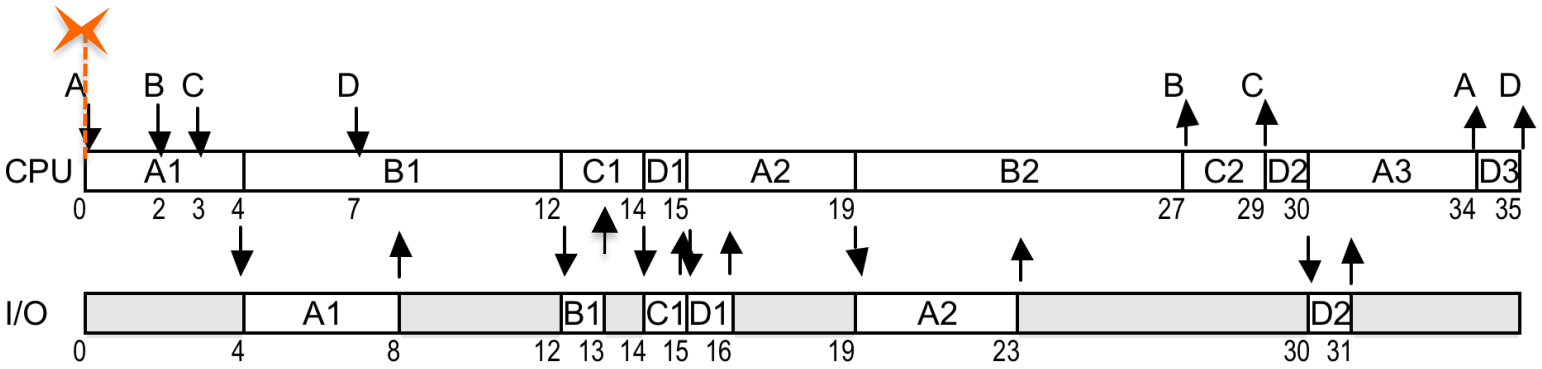
$wt_B = (4 - 2) + (19 - 13) = 8$

$wt_C = (12 - 3) + (27 - 15) = 21$

$wt_D = (14 - 7) + (29 - 16) + (34 - 31) = 23$

$wt_{AVG} = (14 + 12 + 21 + 23) / 4 = 16.5$

First-Come-First-Served (FCFS)



- Response time:

  $rt_A = 0 - 0 = 0$

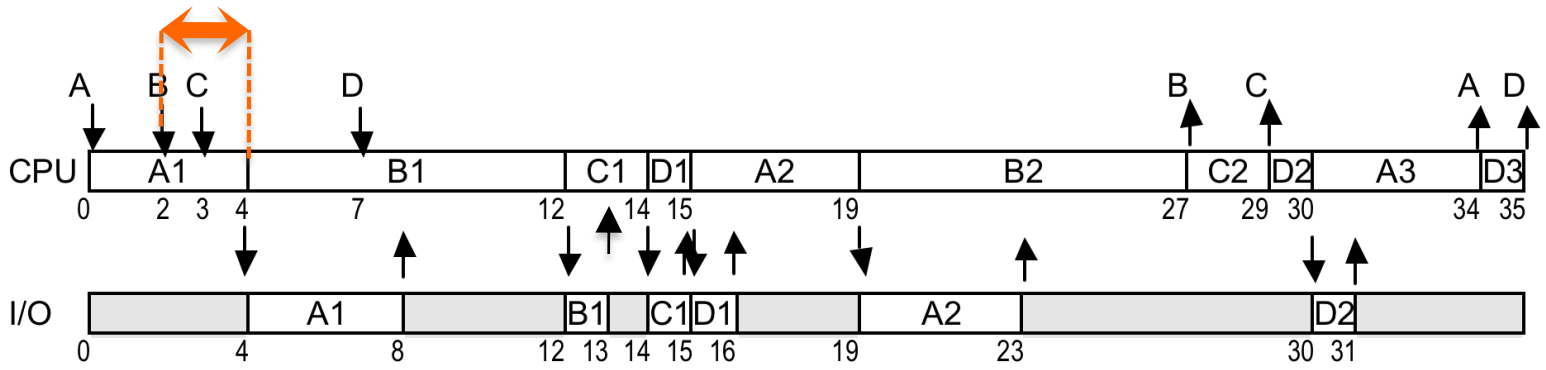
  $rt_B = 4 - 2 = 2$

  $rt_C = 12 - 3 = 9$

  $rt_D = 14 - 7 = 7$

  $rt_{AVG} = (0 + 2 + 9 + 7) / 4 = 4.5$

First-Come-First-Served (FCFS)



- Response time:

$rt_A = 0 - 0 = 0$

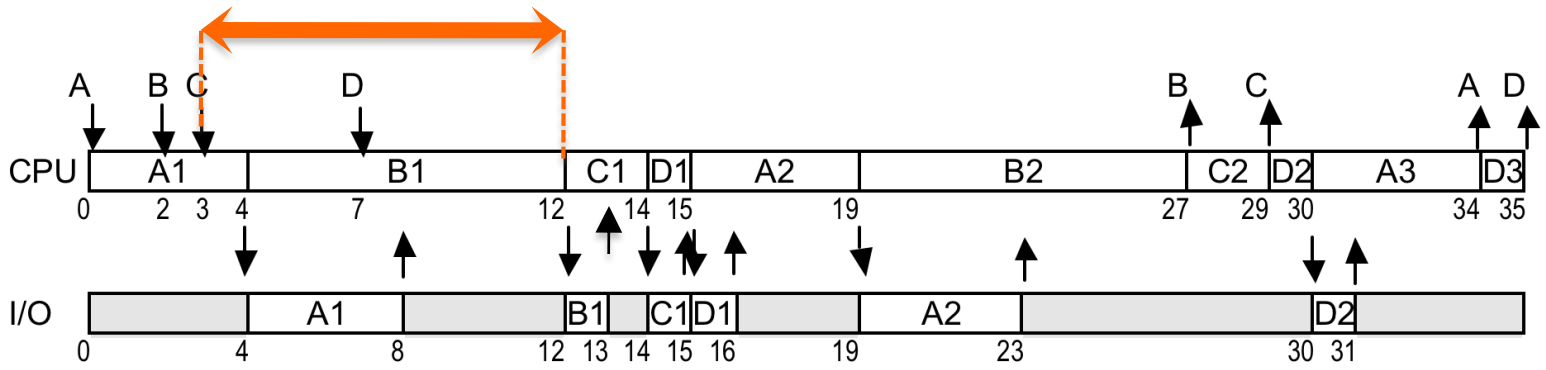
$rt_B = 4 - 2 = 2$

$rt_C = 12 - 3 = 9$

$rt_D = 14 - 7 = 7$

$rt_{AVG} = (0 + 2 + 9 + 7) / 4 = 4.5$

First-Come-First-Served (FCFS)



- Response time:

  $rt_A = 0 - 0 = 0$

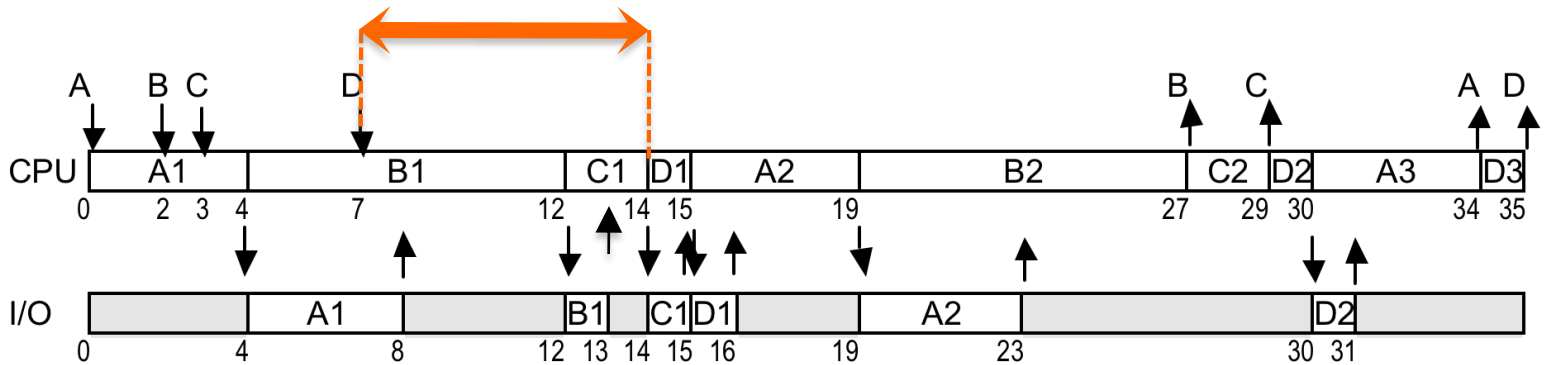
  $rt_B = 4 - 2 = 2$

  $rt_C = 12 - 3 = 9$

  $rt_D = 14 - 7 = 7$

  $rt_{AVG} = (0 + 2 + 9 + 7) / 4 = 4.5$

First-Come-First-Served (FCFS)



- Response time:

  $rt_A = 0 - 0 = 0$

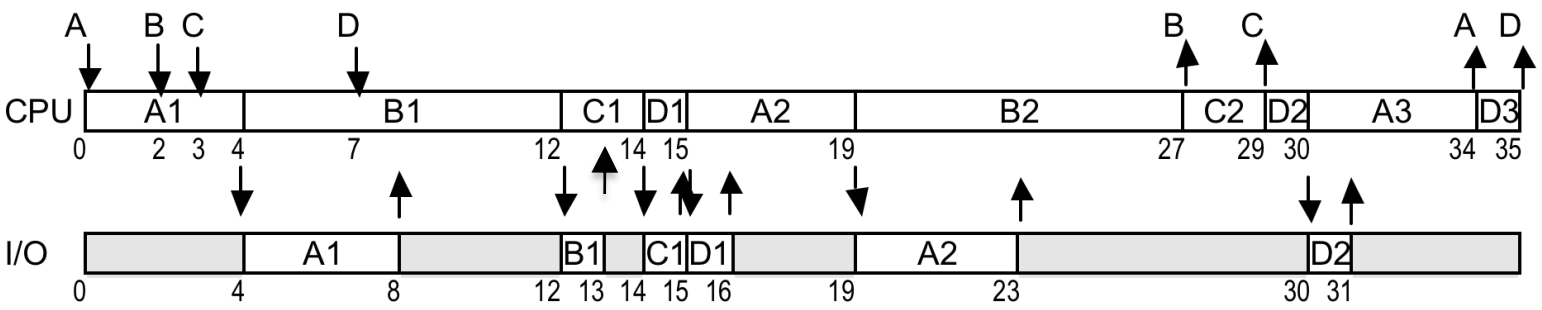
  $rt_B = 4 - 2 = 2$

  $rt_C = 12 - 3 = 9$

  $rt_D = 14 - 7 = 7$

  $rt_{AVG} = (0 + 2 + 9 + 7) / 4 = 4.5$

First-Come-First-Served (FCFS)



- Response time:

  $rt_A = 0 - 0 = 0$

  $rt_B = 4 - 2 = 2$

  $rt_C = 12 - 3 = 9$

  $rt_D = 14 - 7 = 7$

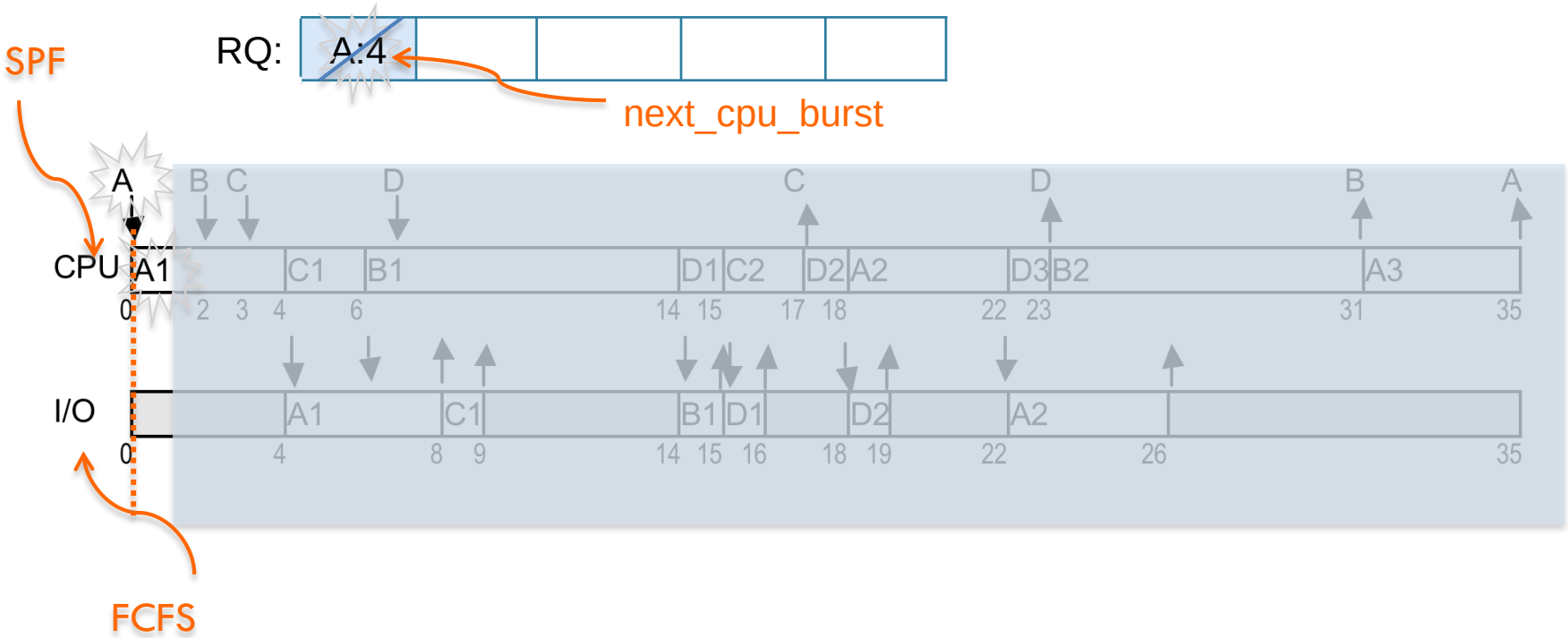
  $rt_{AVG} = (0 + 2 + 9 + 7) / 4 = 4.5$

Shortest-Process-First (SPF)

- In this method, the processor is assigned to *the process with the smallest execution (processor burst) time.*
- This requires future knowledge of the execution time.
- In our examples, it is given as a table but actually these burst times are not known by the OS. So it makes prediction.
- One approach for this prediction is using the previous processor burst times for the processes in the ready queue and then the algorithm selects the shortest predicted next processor burst time.

Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

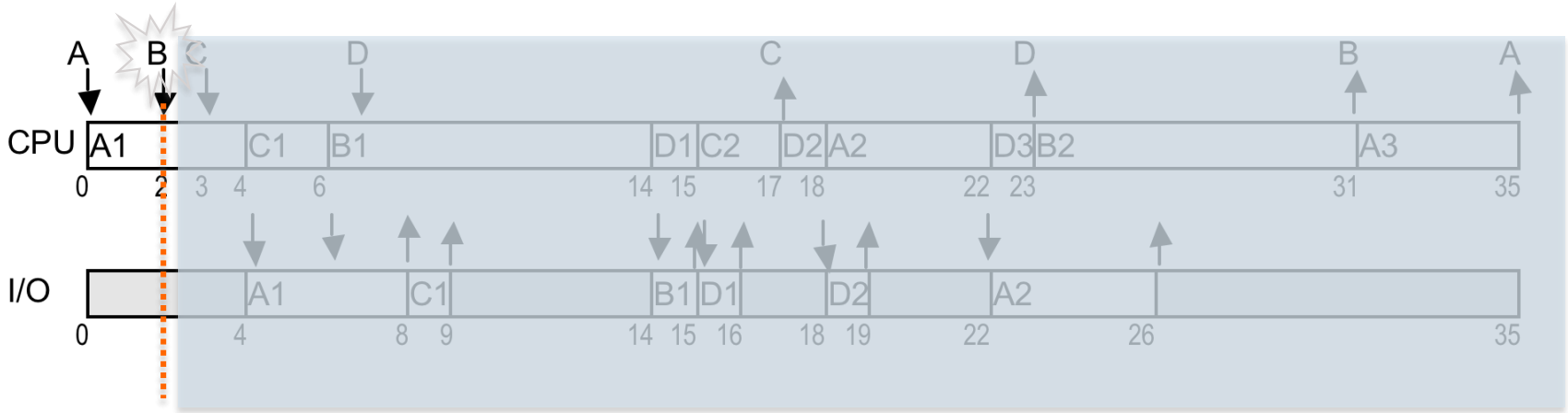


Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: B:8

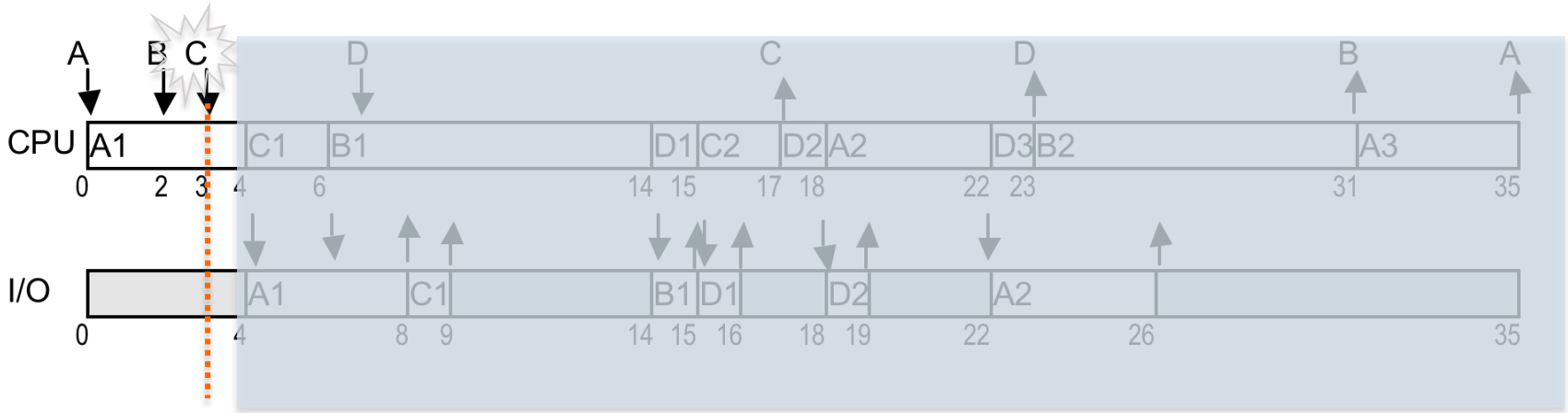
Proc_id : next_cpu_burst



Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

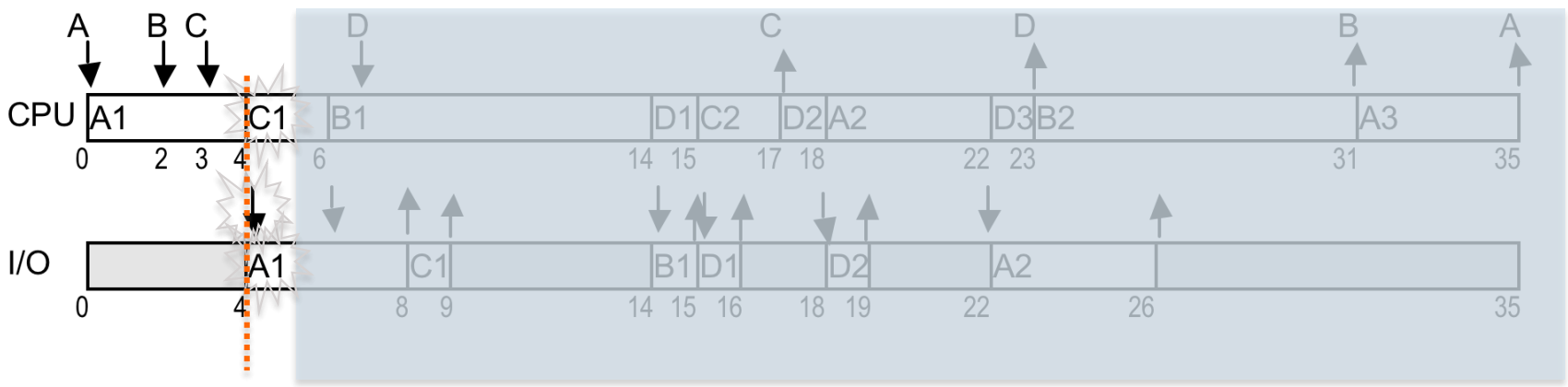
RQ: B:8 | C:2



Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

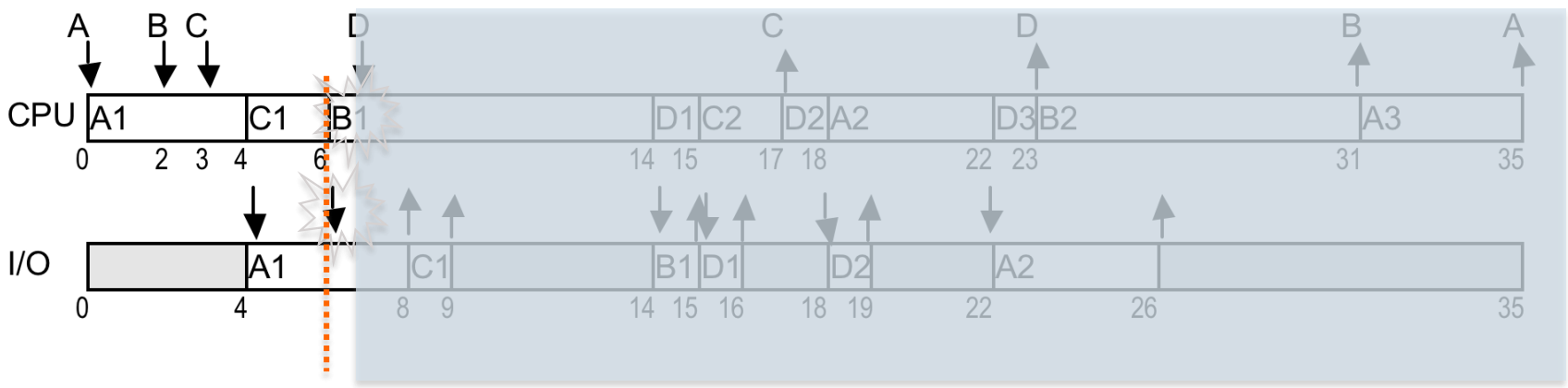
RQ: B:8 | ~~C:2~~ | | | |



Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

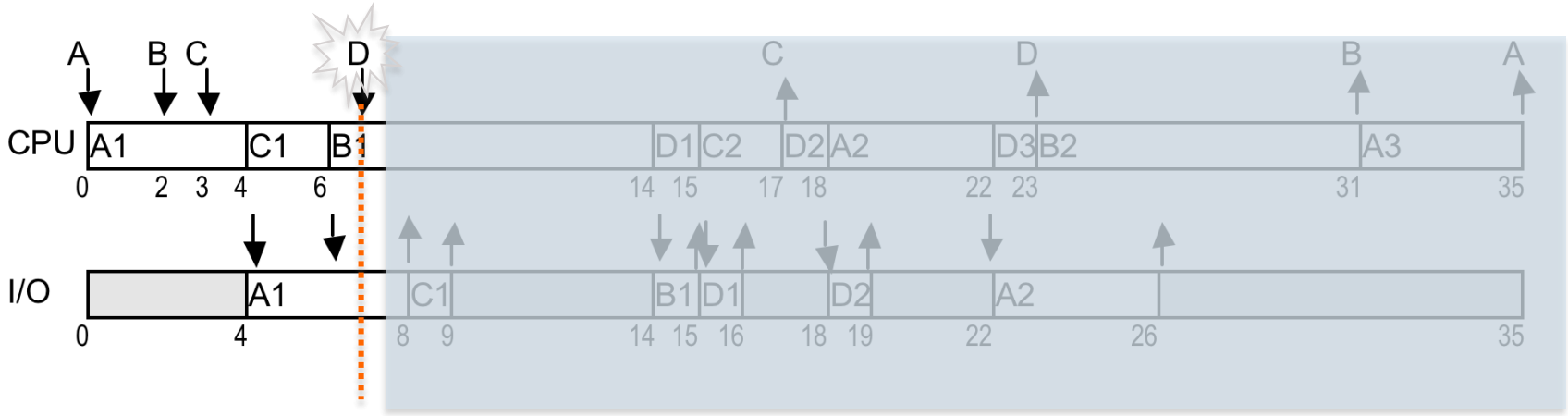
RQ: ~~B.8~~



Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: D:1

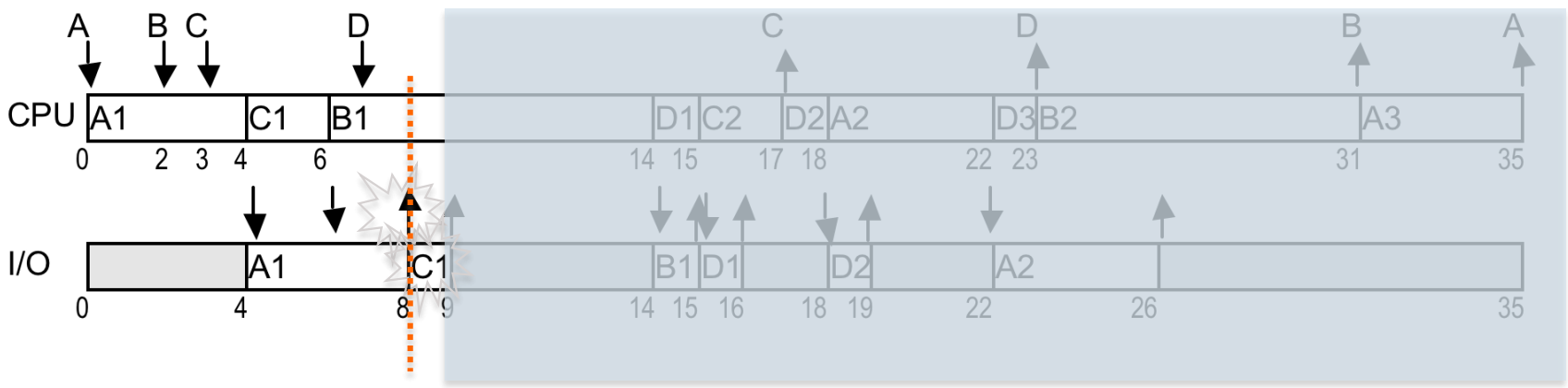


Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ:

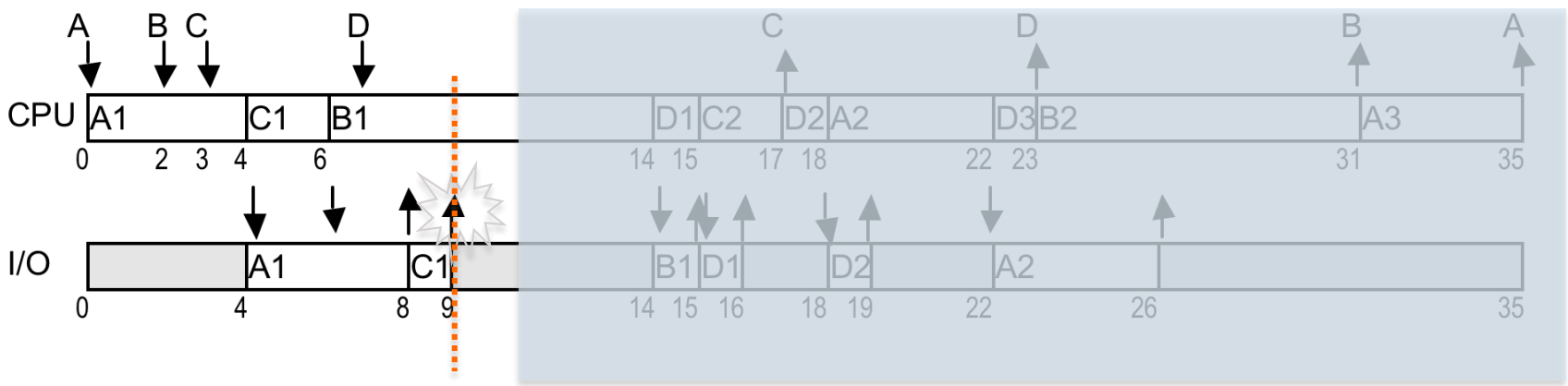
D:1	A:4			



Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

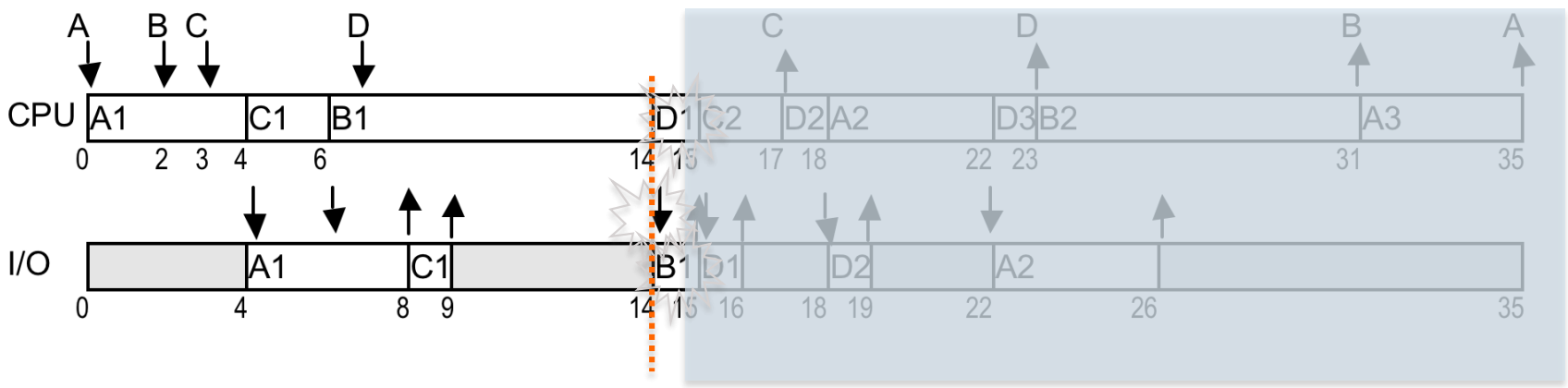
RQ: D:1 | A:4 | C:2



Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

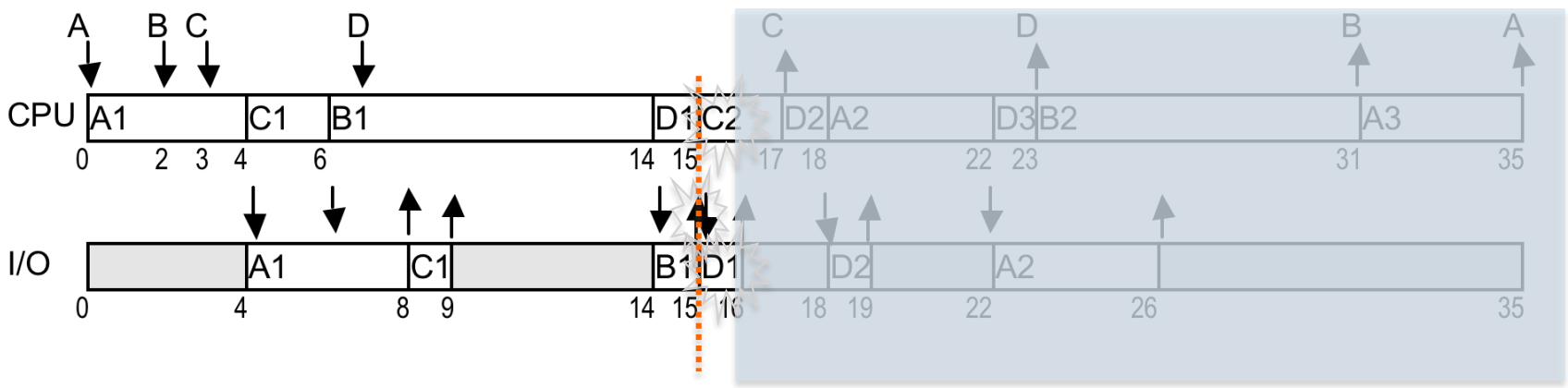
RQ: | ~~D:1~~ | A:4 | C:2 | | |



Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ: A:4 | ~~C:2~~ | B:8 | | |

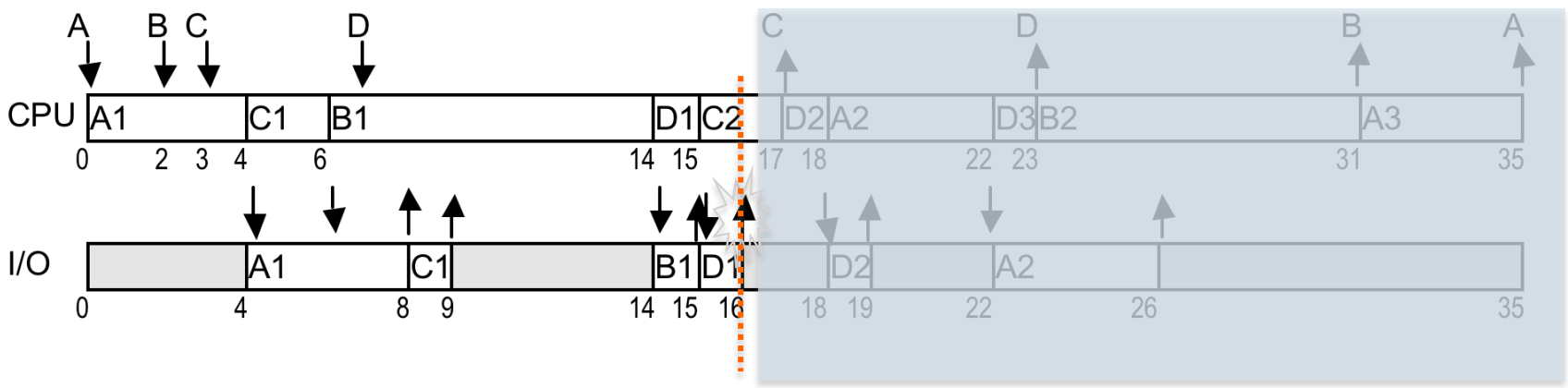


Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ:

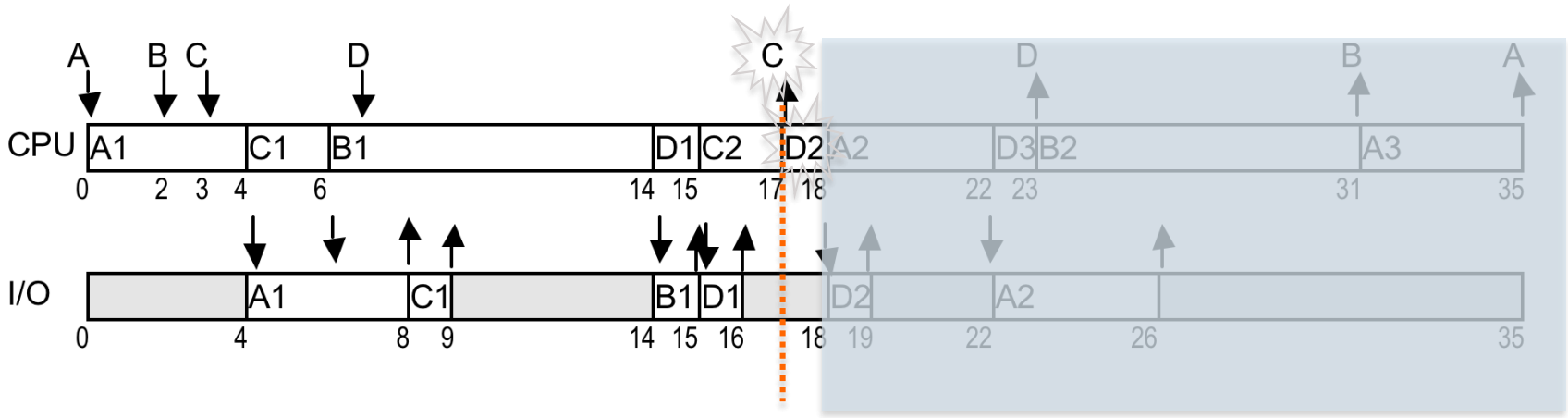
A:4	B:8	D:1		



Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

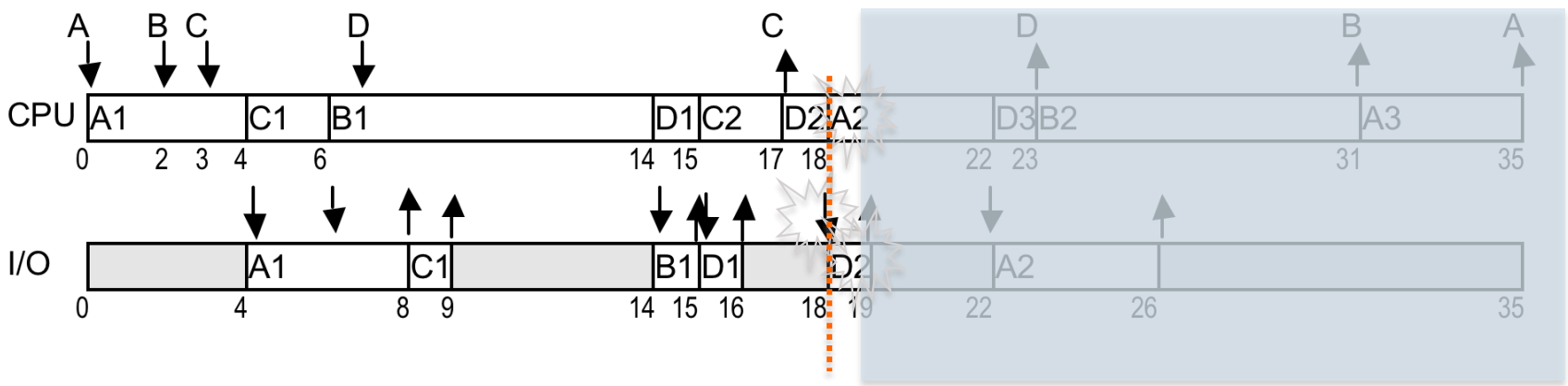
RQ: A:4 | B:8 | ~~D:1~~



Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

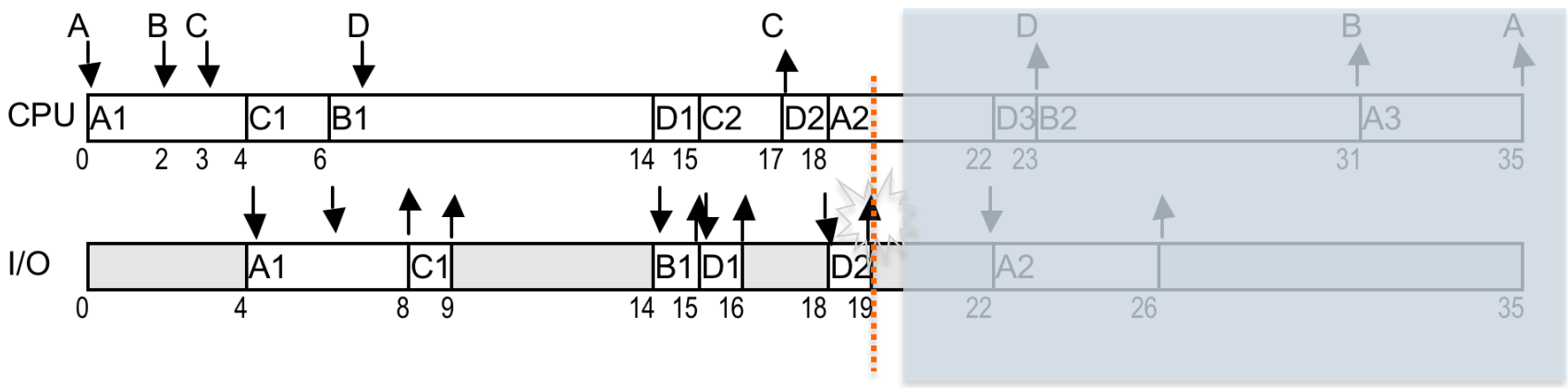
RQ: ~~A:4~~ | B:8 | | | |



Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

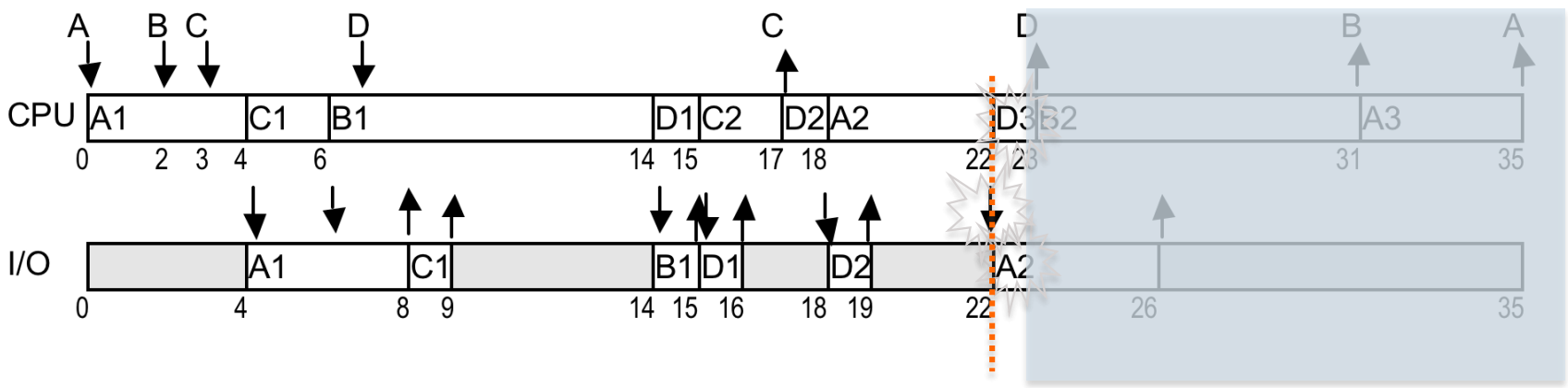
RQ: B:8 | D:1



Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

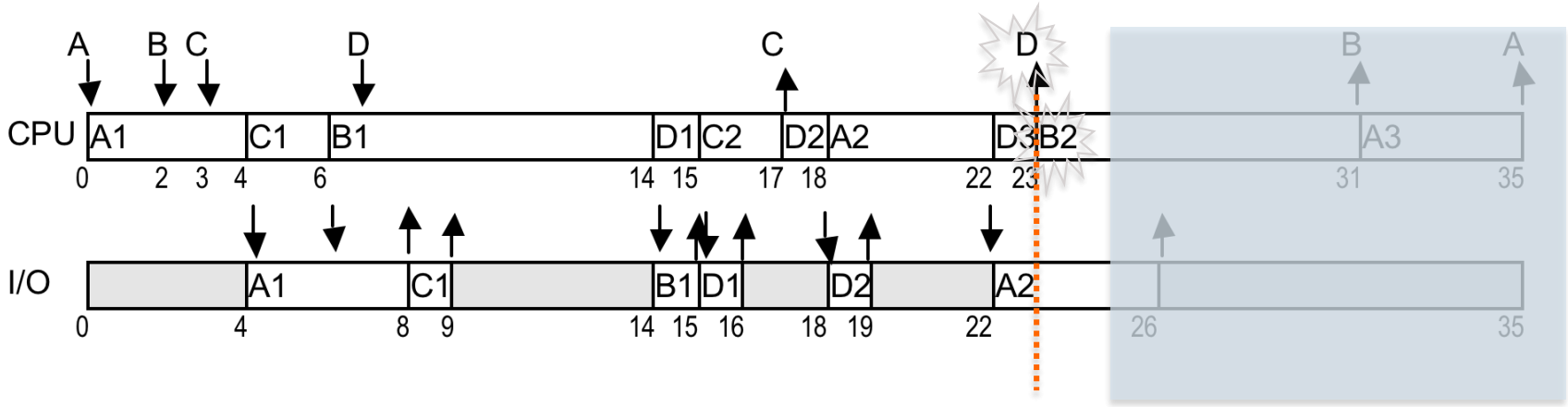
RQ: B:8 | ~~D:1~~ | | | |



Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

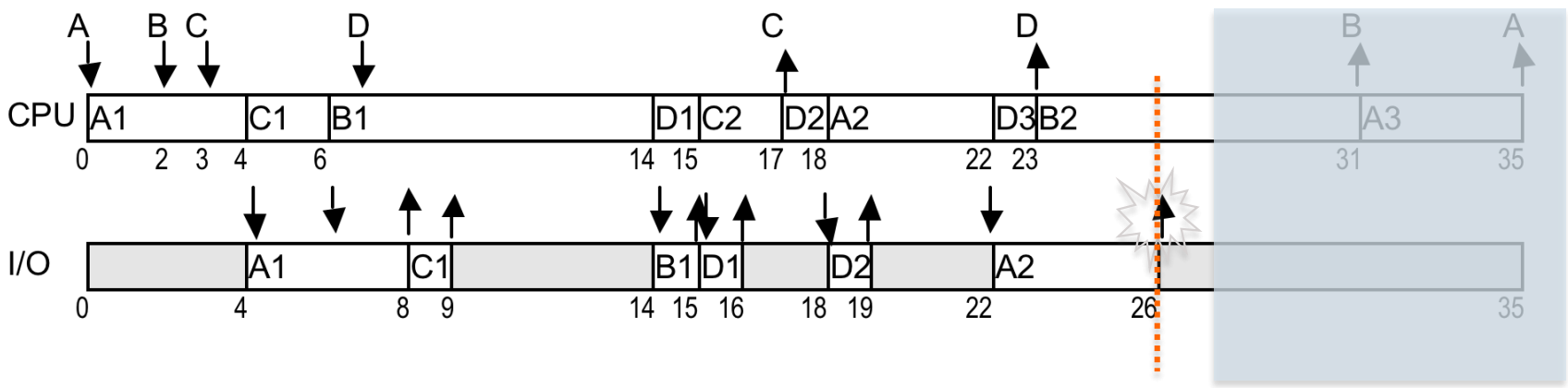
RQ: ~~B:8~~



Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

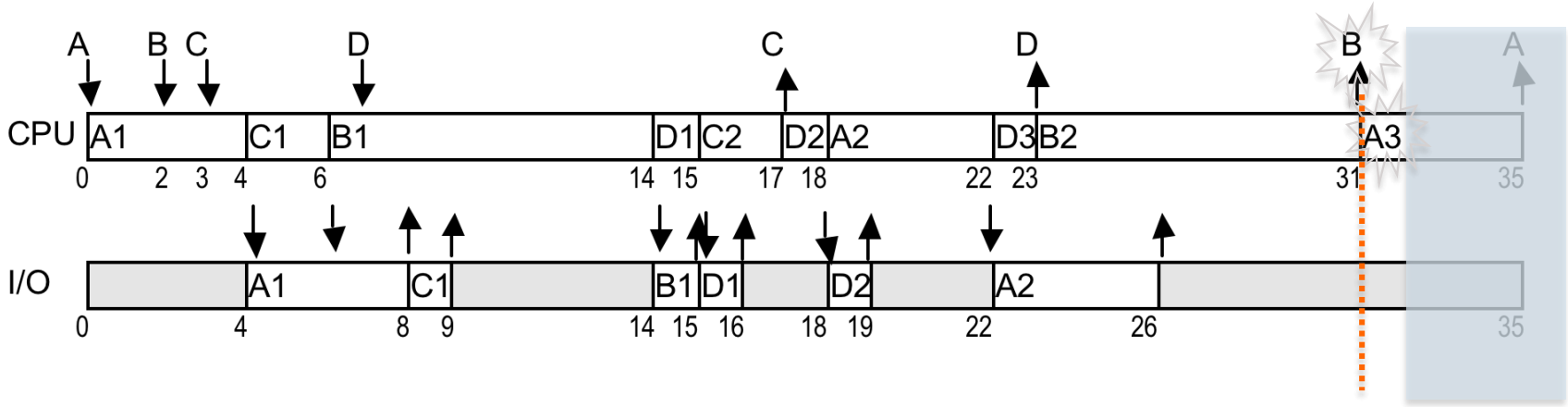
RQ: A:4



Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

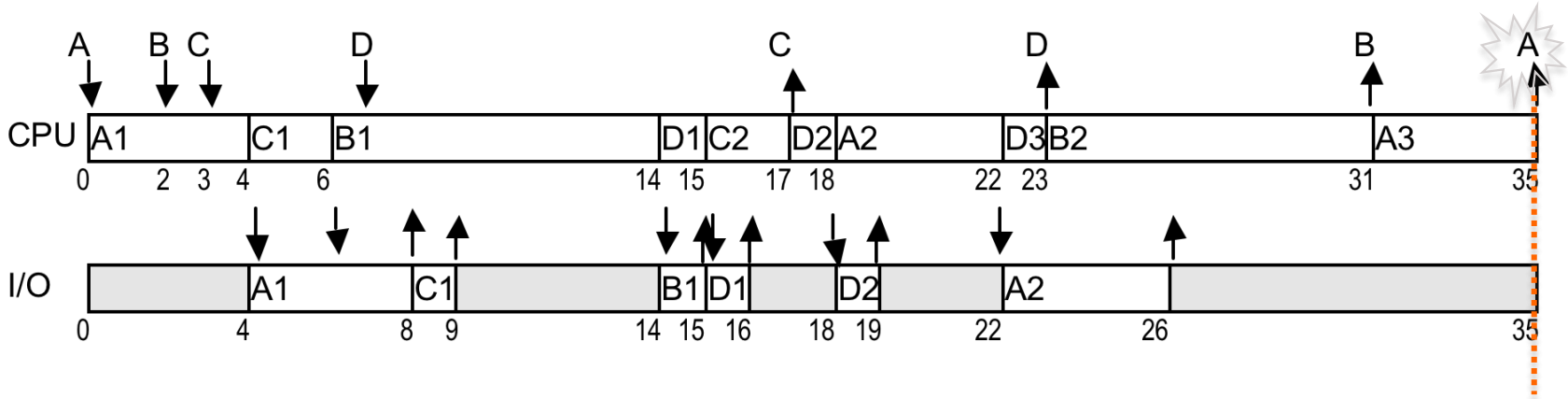
RQ: ~~A.4~~



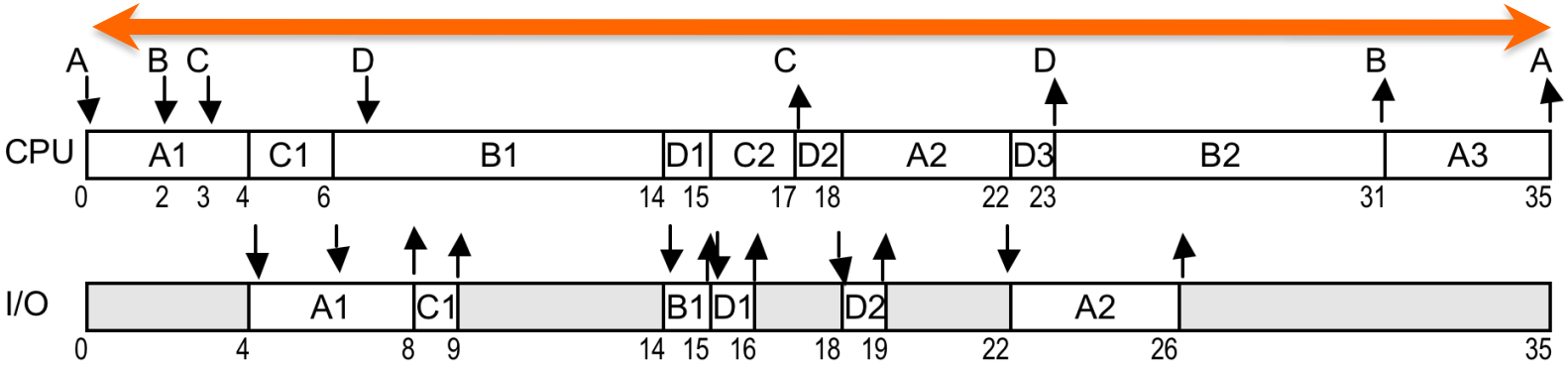
Shortest-Process-First (SPF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

RQ:

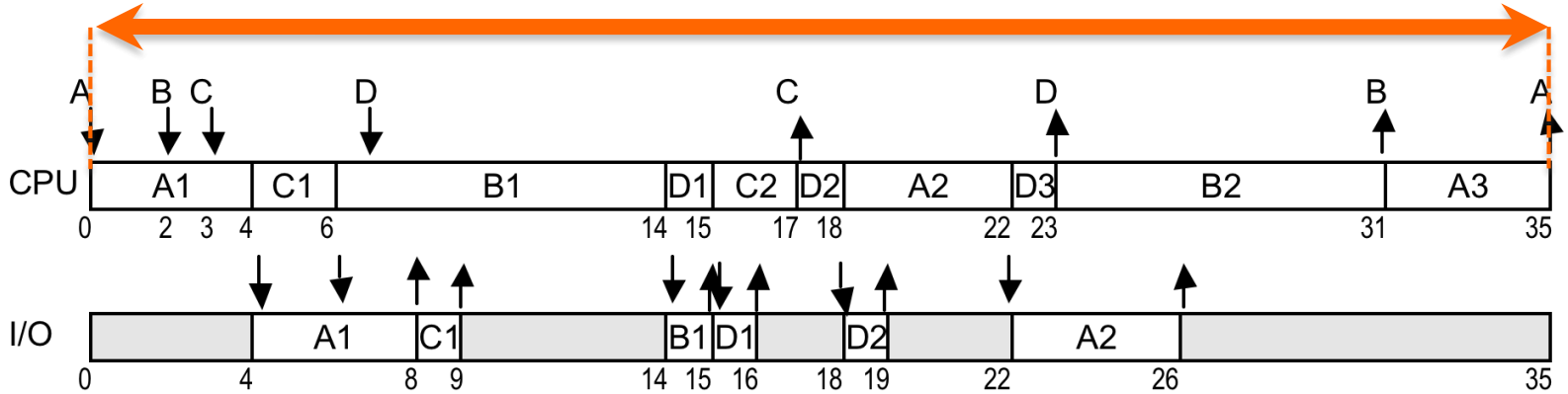


Shortest-Process-First (SPF)



- Processor utilization = (35 / 35) * 100 = 100 %
- Throughput = 4 / 35 = 0.11

Shortest-Process-First (SPF)



- Turn around time:

  $tat_A = 35 - 0 = 35$

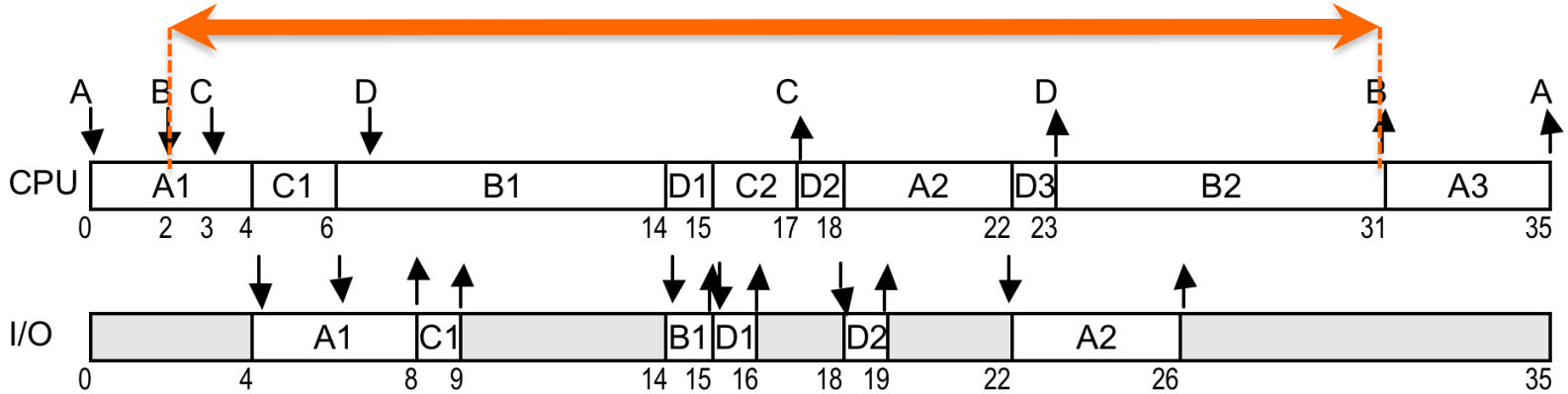
  $tat_B = 31 - 2 = 29$

  $tat_C = 17 - 3 = 14$

  $tat_D = 23 - 7 = 16$

  $tat_{AVG} = (35 + 29 + 15 + 16) / 4 = 23.5$

Shortest-Process-First (SPF)



- Turn around time:

  $tat_A = 35 - 0 = 35$

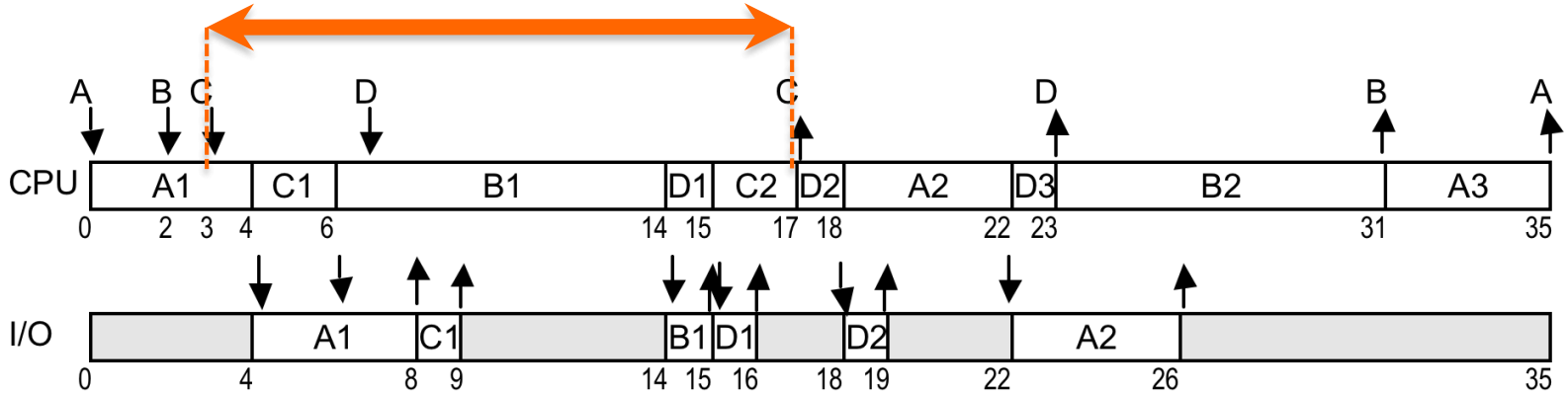
  $tat_B = 31 - 2 = 29$

  $tat_C = 17 - 3 = 14$

  $tat_D = 23 - 7 = 16$

  $tat_{AVG} = (35 + 29 + 15 + 16) / 4 = 23.5$

Shortest-Process-First (SPF)



- Turn around time:

  $tat_A = 35 - 0 = 35$

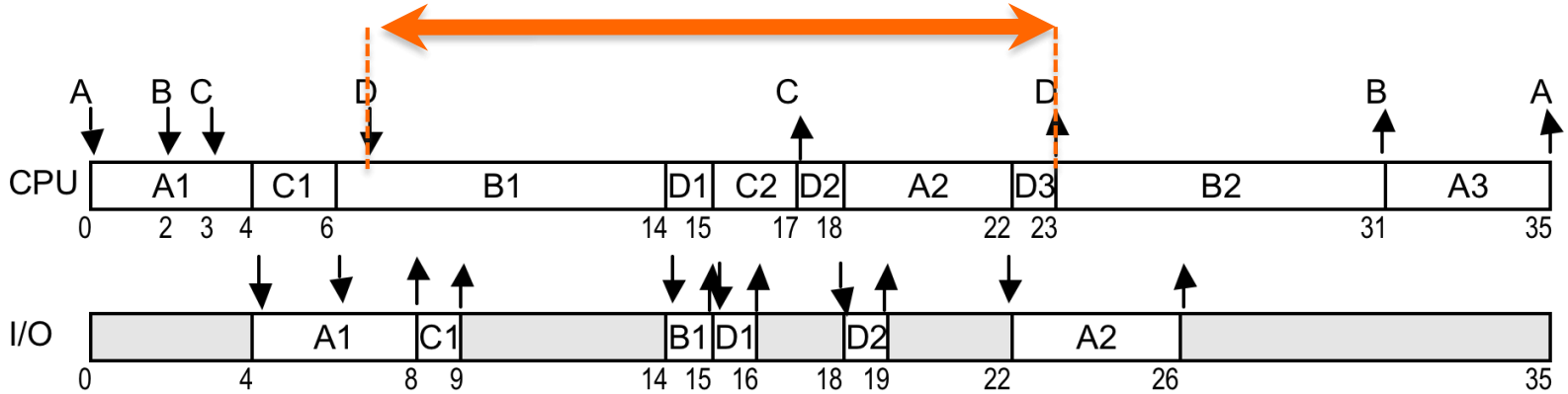
  $tat_B = 31 - 2 = 29$

  $tat_C = 17 - 3 = 14$

  $tat_D = 23 - 7 = 16$

  $tat_{AVG} = (35 + 29 + 15 + 16) / 4 = 23.5$

Shortest-Process-First (SPF)



- Turn around time:

  $tat_A = 35 - 0 = 35$

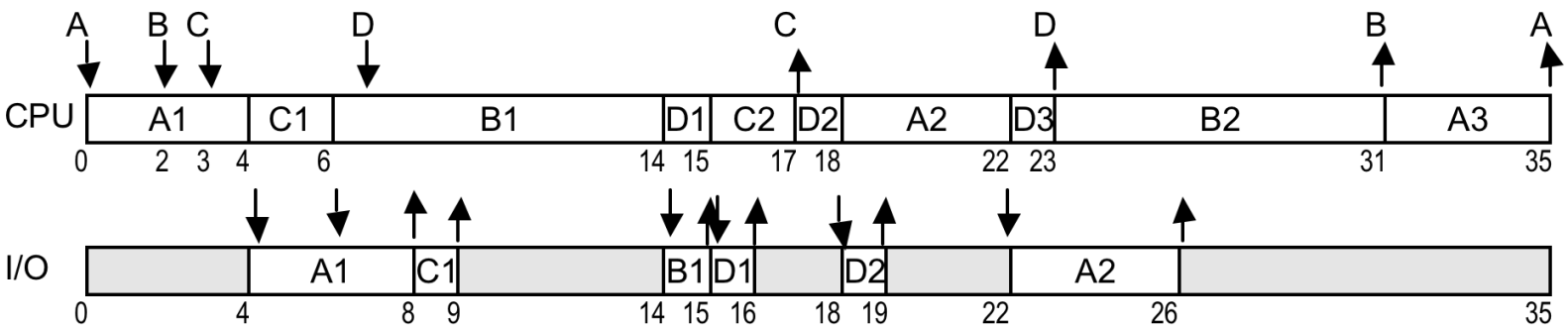
  $tat_B = 31 - 2 = 29$

  $tat_C = 17 - 3 = 14$

  $tat_D = 23 - 7 = 16$

  $tat_{AVG} = (35 + 29 + 15 + 16) / 4 = 23.5$

Shortest-Process-First (SPF)



- Turn around time:

  $tat_A = 35 - 0 = 35$

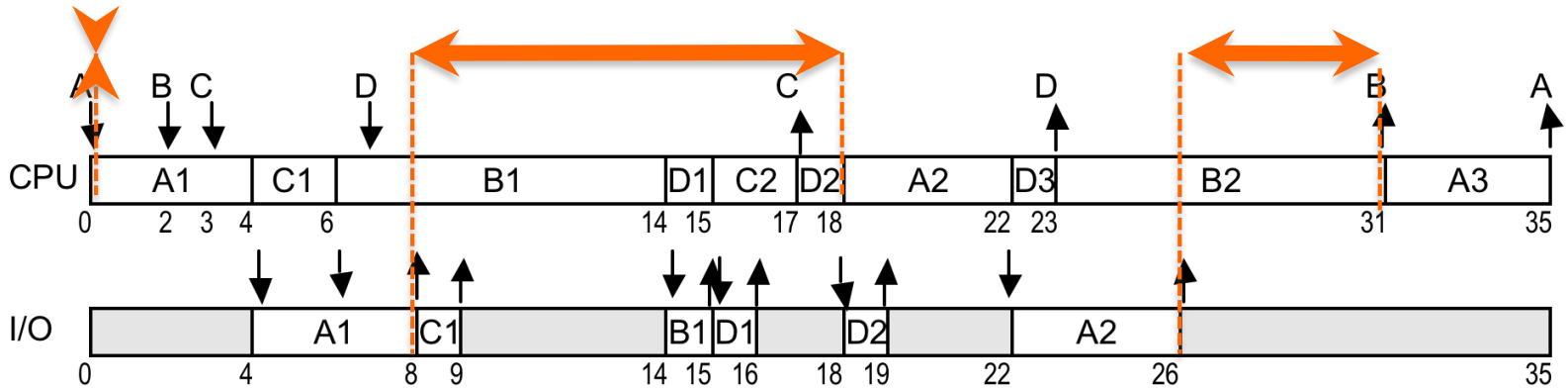
  $tat_B = 31 - 2 = 29$

  $tat_C = 17 - 3 = 14$

  $tat_D = 23 - 7 = 16$

  $tat_{AVG} = (35 + 29 + 15 + 16) / 4 = 23.5$

Shortest-Process-First (SPF)



- Waiting time:

$wt_A = (0 - 0) + (18 - 8) + (31 - 26) = 15$

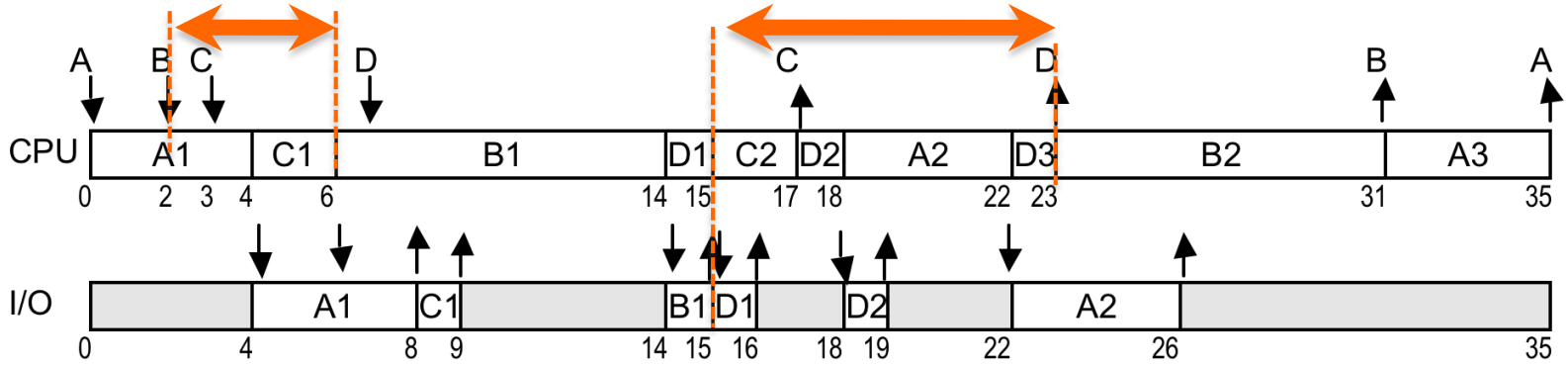
$wt_B = (6 - 2) + (23 - 15) = 12$

$wt_C = (4 - 3) + (15 - 9) = 7$

$wt_D = (14 - 7) + (17 - 16) + (22 - 19) = 11$

$wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

Shortest-Process-First (SPF)



- Waiting time:

  $wt_A = (0 - 0) + (18 - 8) + (31 - 26) = 15$

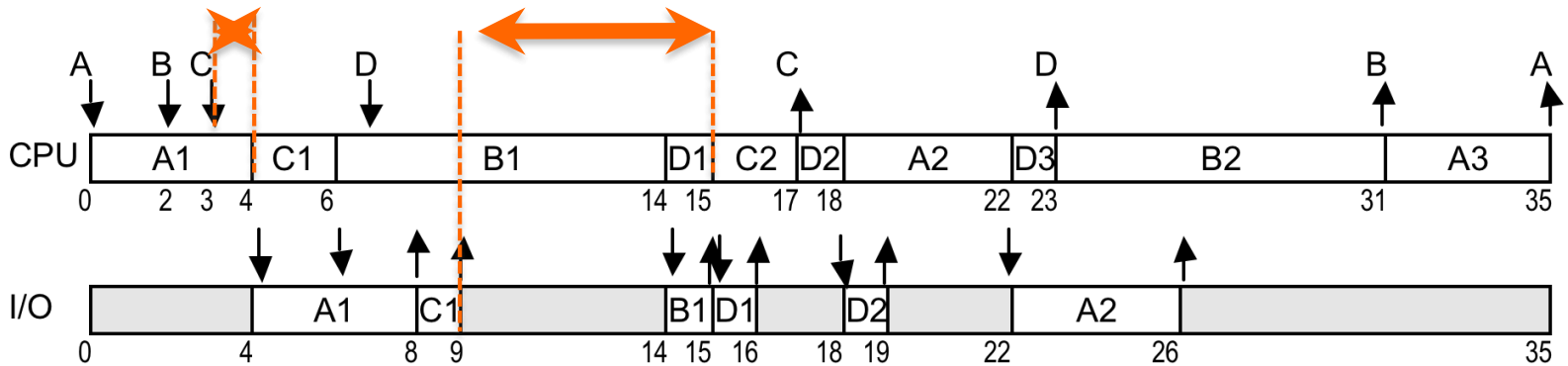
  $wt_B = (6 - 2) + (23 - 15) = 12$

  $wt_C = (4 - 3) + (15 - 9) = 7$

  $wt_D = (14 - 7) + (17 - 16) + (22 - 19) = 11$

  $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

Shortest-Process-First (SPF)



- Waiting time:

  $wt_A = (0 - 0) + (18 - 8) + (31 - 26) = 15$

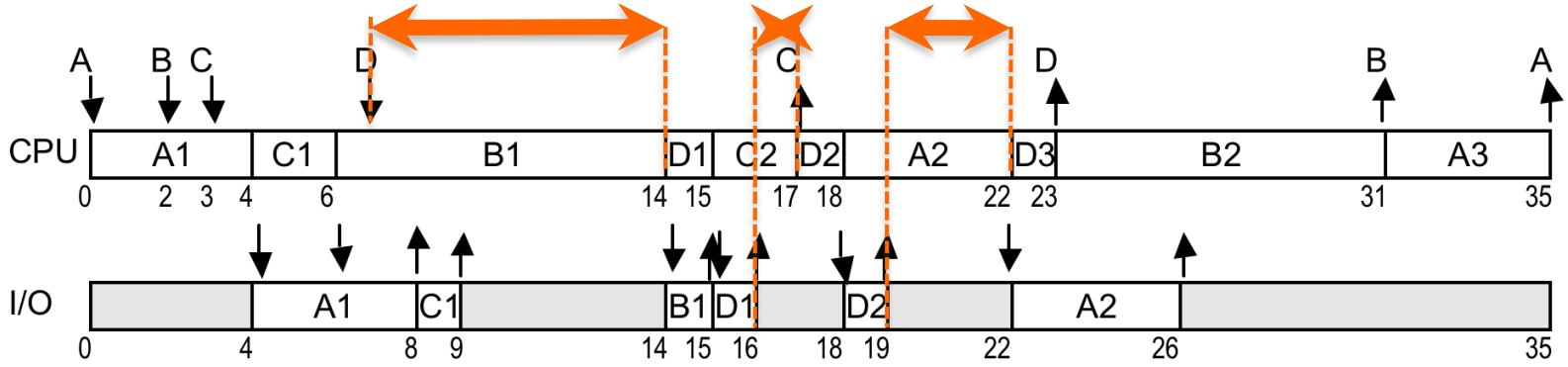
  $wt_B = (6 - 2) + (23 - 15) = 12$

  $wt_C = (4 - 3) + (15 - 9) = 7$

  $wt_D = (14 - 7) + (17 - 16) + (22 - 19) = 11$

  $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

Shortest-Process-First (SPF)



- Waiting time:

  $wt_A = (0 - 0) + (18 - 8) + (31 - 26) = 15$

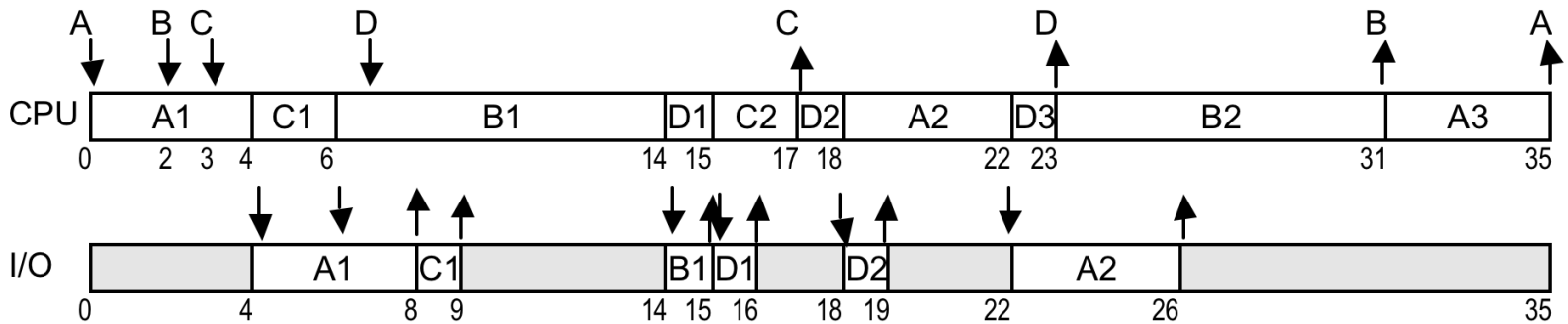
  $wt_B = (6 - 2) + (23 - 15) = 12$

  $wt_C = (4 - 3) + (15 - 9) = 7$

  $wt_D = (14 - 7) + (17 - 16) + (22 - 19) = 11$

  $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

Shortest-Process-First (SPF)



- Waiting time:

  $wt_A = (0 - 0) + (18 - 8) + (31 - 26) = 15$

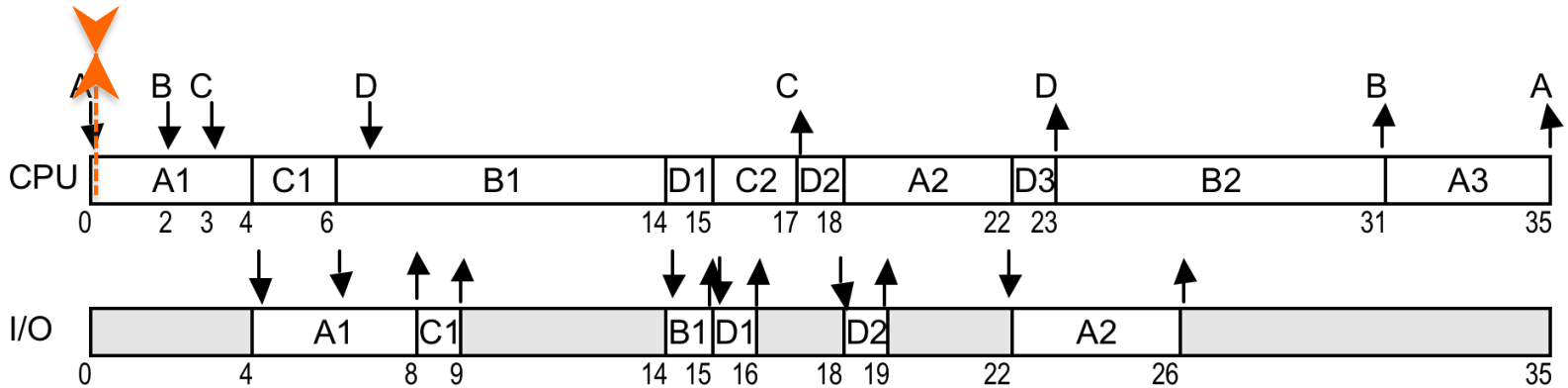
  $wt_B = (6 - 2) + (23 - 15) = 12$

  $wt_C = (4 - 3) + (15 - 9) = 7$

  $wt_D = (14 - 7) + (17 - 16) + (22 - 19) = 11$

  $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

Shortest-Process-First (SPF)



- Response time:

  $rt_A = 0 - 0 = 0$

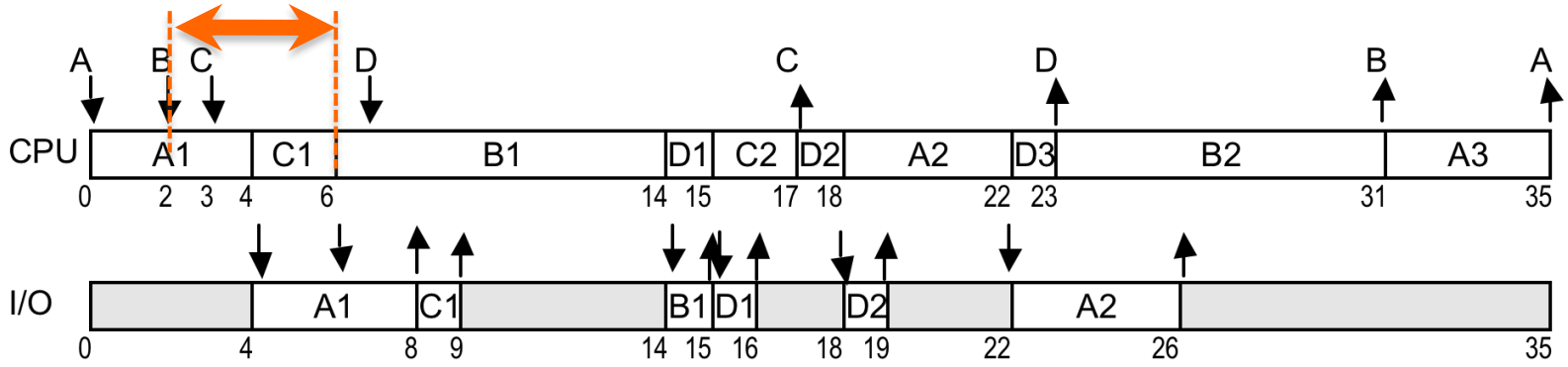
  $rt_B = 6 - 2 = 4$

  $rt_C = 4 - 3 = 1$

  $rt_D = 14 - 7 = 7$

  $rt_{AVG} = (0 + 4 + 1 + 7) / 4 = 3$

Shortest-Process-First (SPF)



- Response time:

  $rt_A = 0 - 0 = 0$

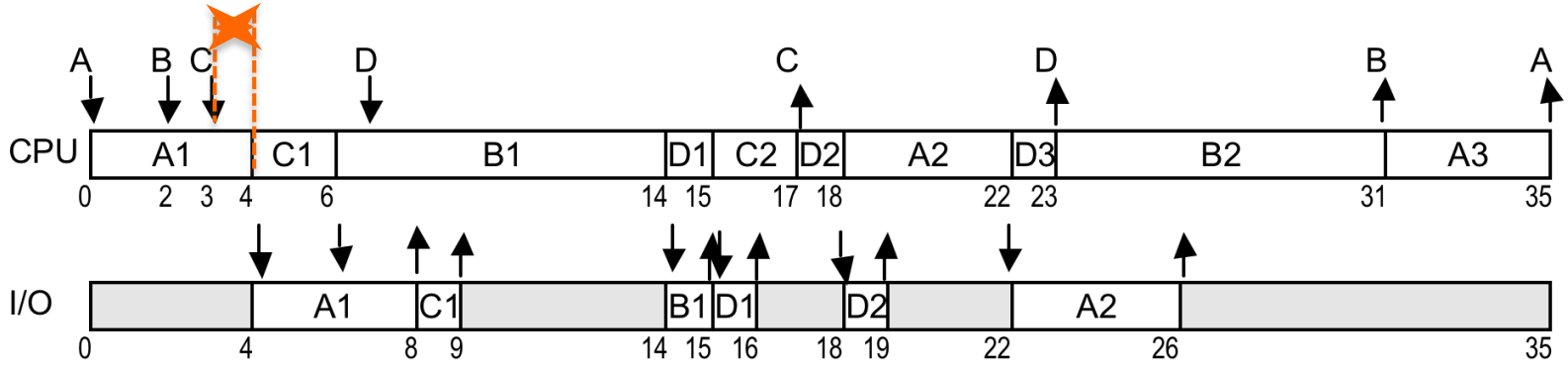
  $rt_B = 6 - 2 = 4$

  $rt_C = 4 - 3 = 1$

  $rt_D = 14 - 7 = 7$

  $rt_{AVG} = (0 + 4 + 1 + 7) / 4 = 3$

Shortest-Process-First (SPF)



- Response time:

  $rt_A = 0 - 0 = 0$

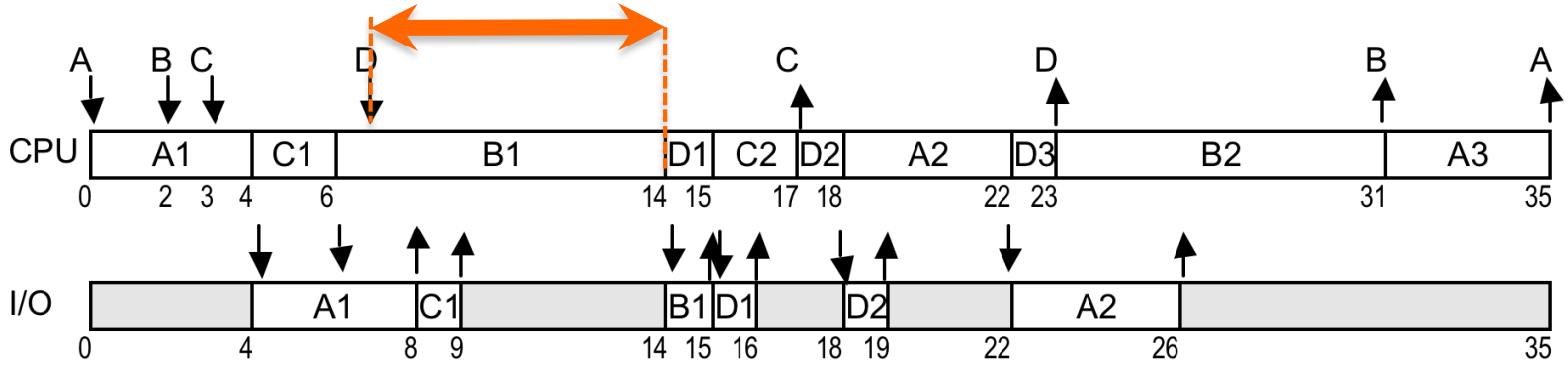
  $rt_B = 6 - 2 = 4$

  $rt_C = 4 - 3 = 1$

  $rt_D = 14 - 7 = 7$

  $rt_{AVG} = (0 + 4 + 1 + 7) / 4 = 3$

Shortest-Process-First (SPF)



- Response time:

  $rt_A = 0 - 0 = 0$

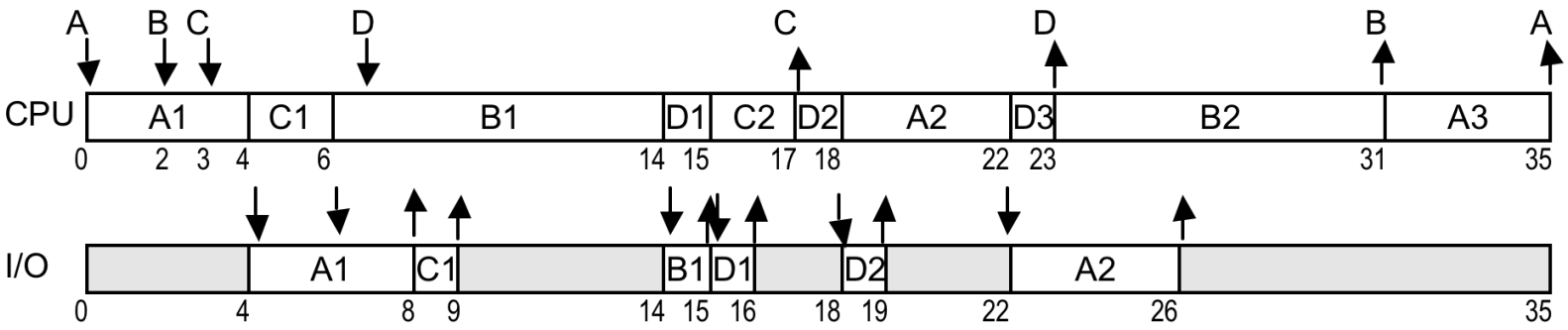
  $rt_B = 6 - 2 = 4$

  $rt_C = 4 - 3 = 1$

  $rt_D = 14 - 7 = 7$

  $rt_{AVG} = (0 + 4 + 1 + 7) / 4 = 3$

Shortest-Process-First (SPF)



- Response time:

  $rt_A = 0 - 0 = 0$

  $rt_B = 6 - 2 = 4$

  $rt_C = 4 - 3 = 1$

  $rt_D = 14 - 7 = 7$

  $rt_{AVG} = (0 + 4 + 1 + 7) / 4 = 3$

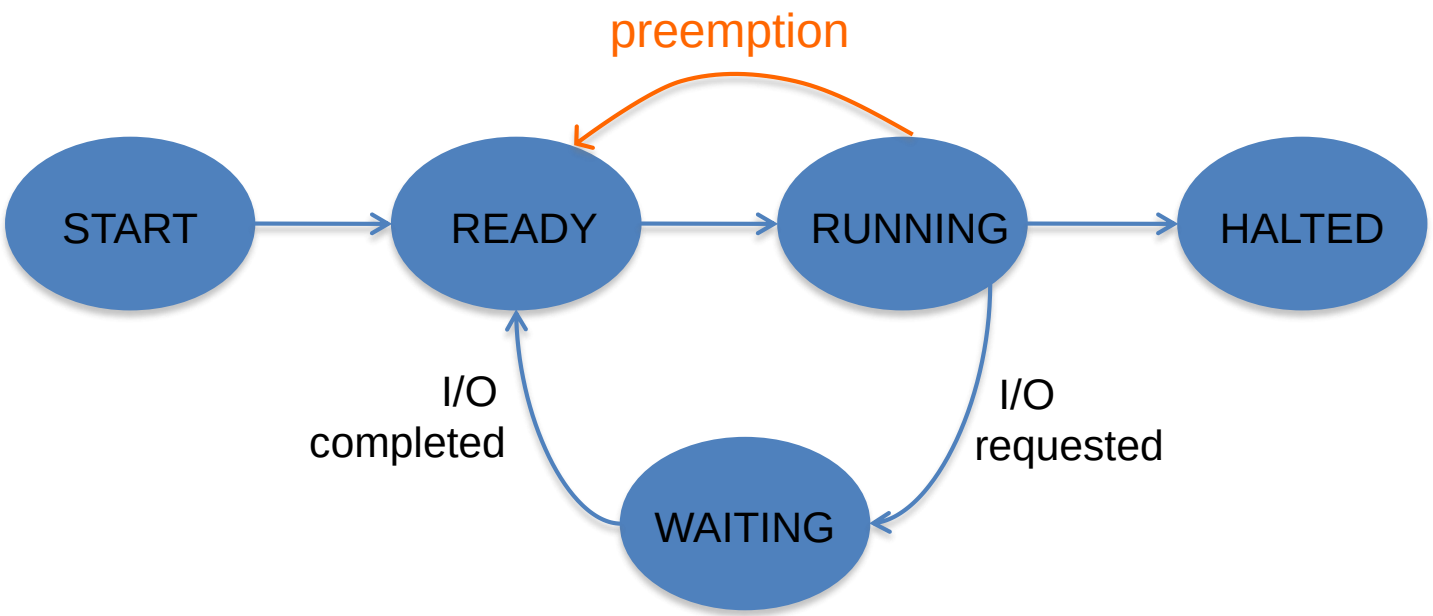
Shortest-Remaining-Time-First (SRTF)

- The scheduling algorithms we discussed so far are all **non-preemptive** algorithms.
- That is, once a process grabs the processor, it keeps the processor until it terminates or it requests I/O.
- To deal with this problem (if so), **preemptive algorithms** are developed.
- In this type of algorithms, at some time instant, the process being executed is forced to preempt CPU in order to execute a new selected process.
- The preemption conditions are up to the algorithm design.

Shortest-Remaining-Time-First (SRTF)

- SPF algorithm can be modified to be preemptive.
- Assume while one process is executing on the processor, another process arrives.
- The new process may have a predicted next processor burst time shorter than what is left of the currently executing process.
- If the SPF algorithm is preemptive, the currently executing process will preempt the processor and the new process will start executing.
- The modified SPF algorithm is named as Preemptive SPF or Shortest-Remaining-Time-First (SRTF) algorithm.

Shortest-Remaining-Time-First (SRTF)



Start : The process has just arrived.
Ready : The process is waiting to grab the processor.
Running : The process has been allocated by the processor.
Waiting : The process is doing I/O work or blocked.
Halted : The process has finished and is about to leave the system

Shortest-Remaining-Time-First (SRTF)

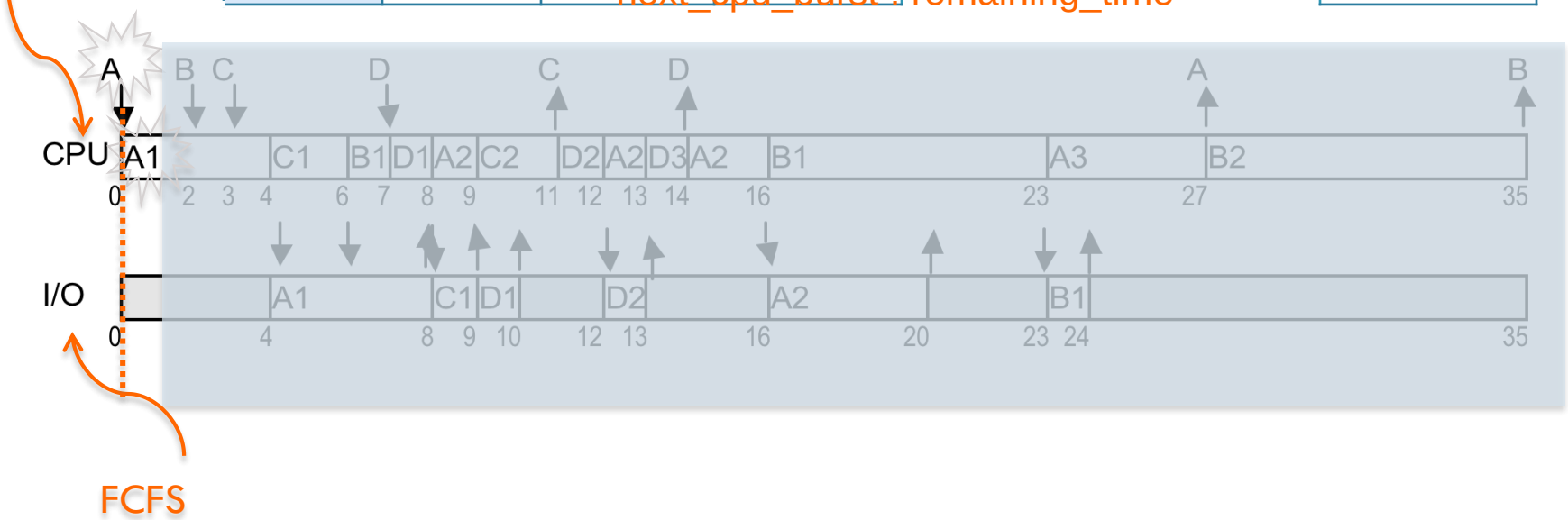
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 4	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

SRTF

RQ : ~~A:4:4~~

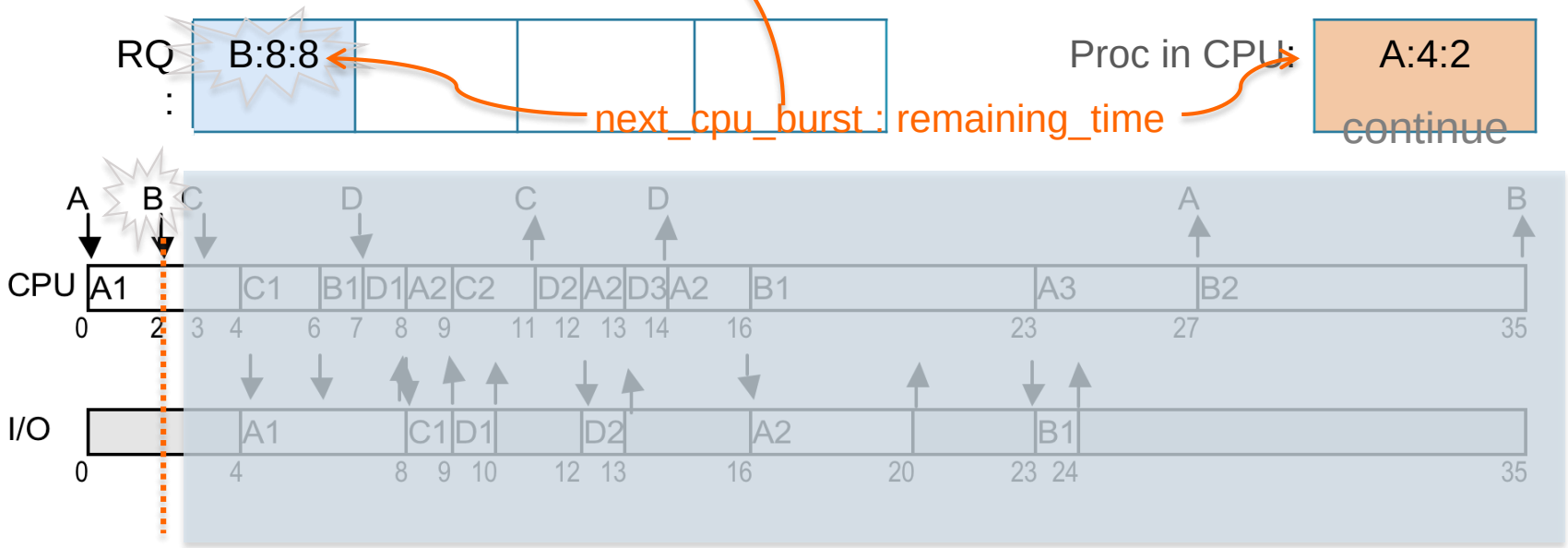
next_cpu_burst : remaining_time

Proc in CPU:



Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 2	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

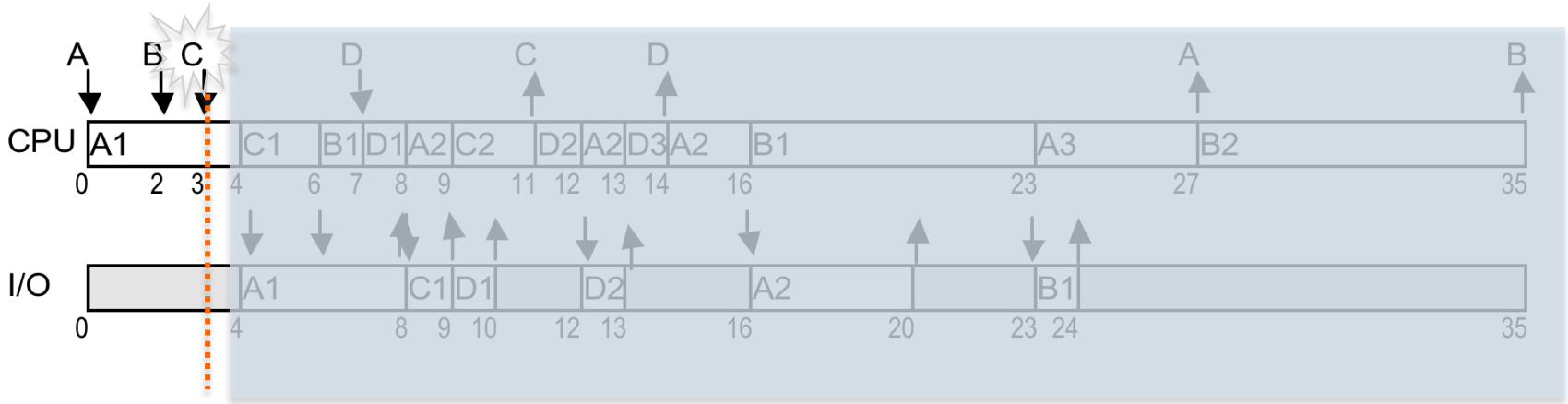


Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 1	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

RQ : B:8:8 | C:2:2

Proc in CPU: A:4:1 continue

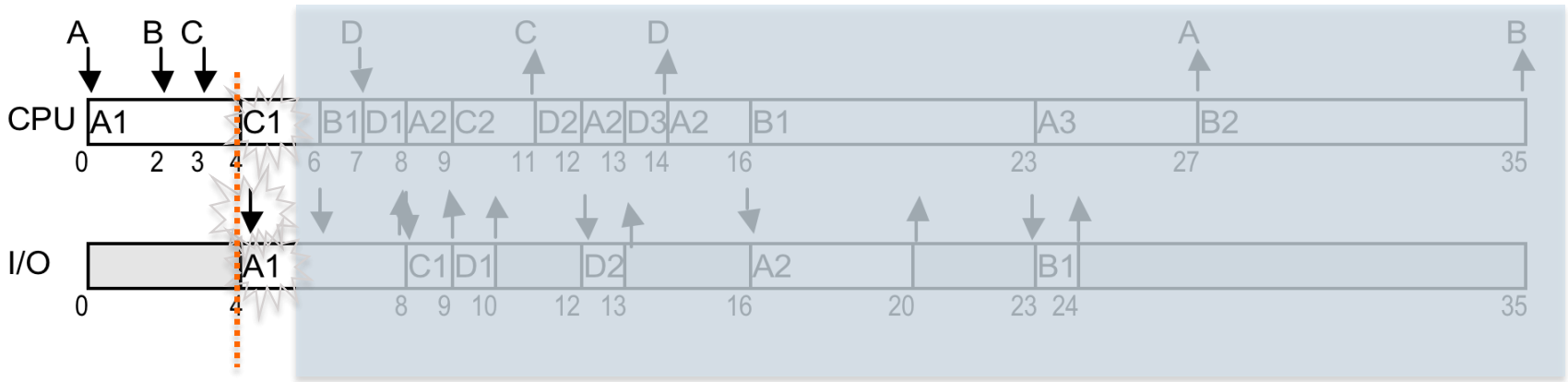


Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

RQ : B:8:8 ~~C:2:2~~

Proc in CPU:

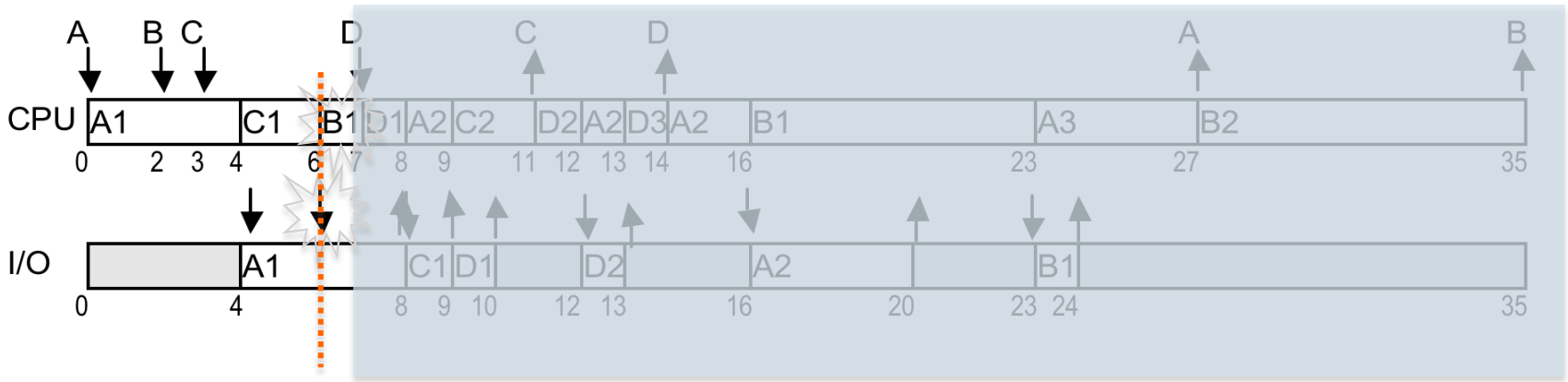


Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

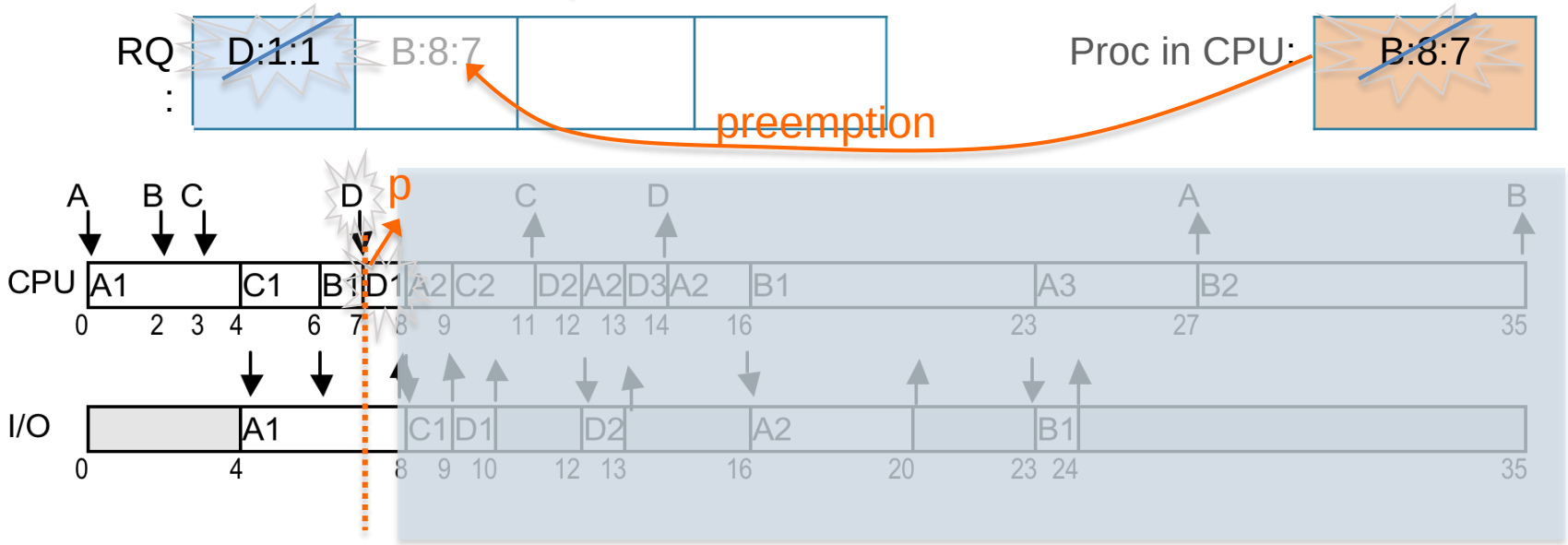
RQ : ~~B:8:8~~

Proc in CPU:



Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1



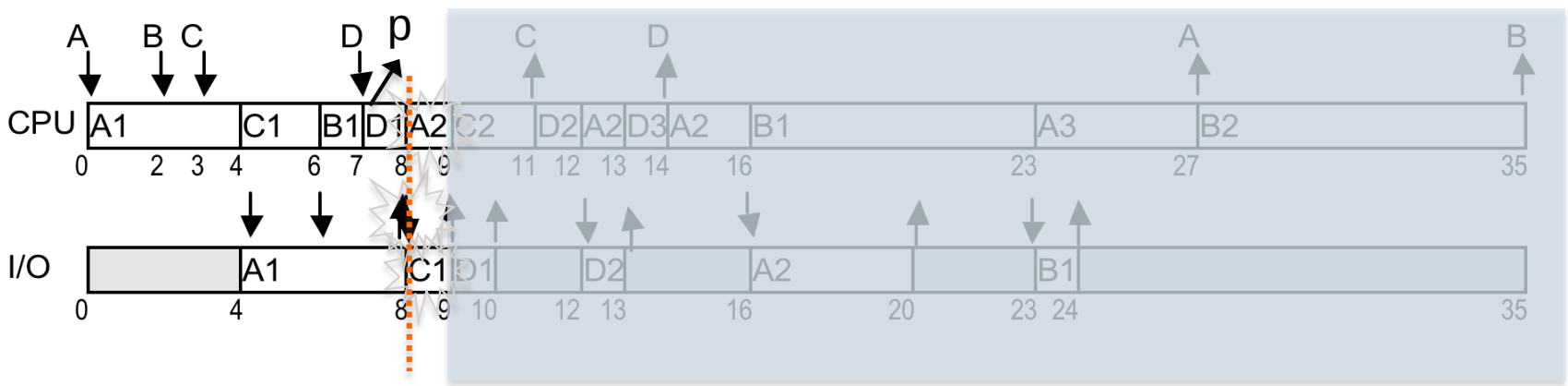
Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

RQ :

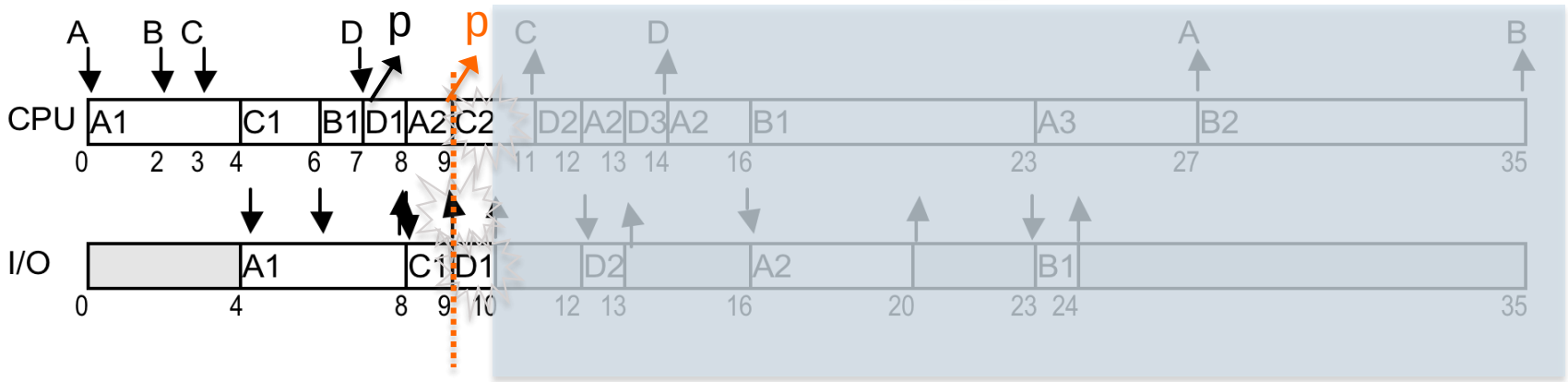
B:8:7	~~A:4:4~~		

Proc in CPU:



Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 3	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1



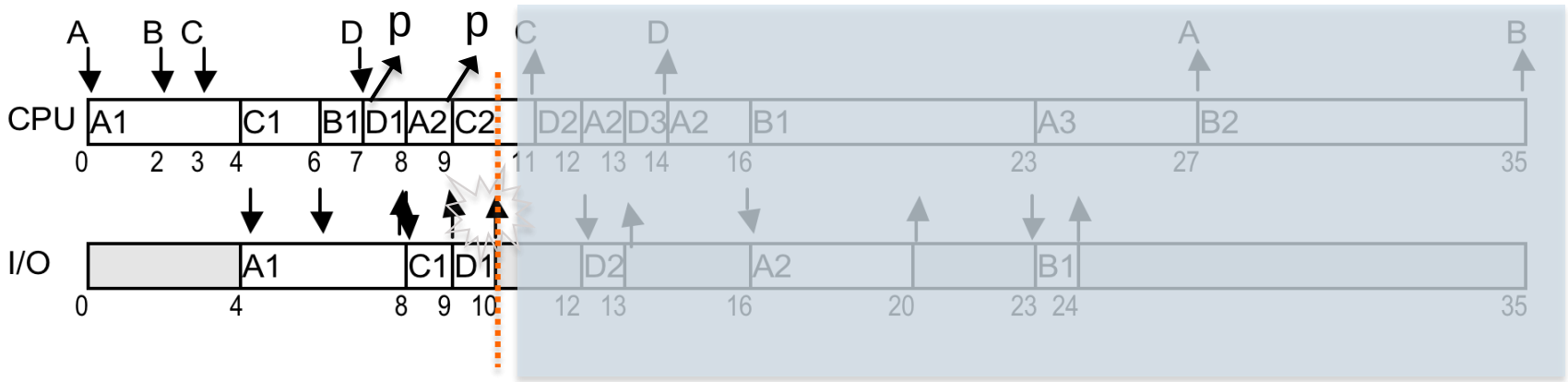
Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 3	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 1	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

RQ :

B:8:7	A:4:3	D:1:1	

Proc in CPU: C:2:1 continues

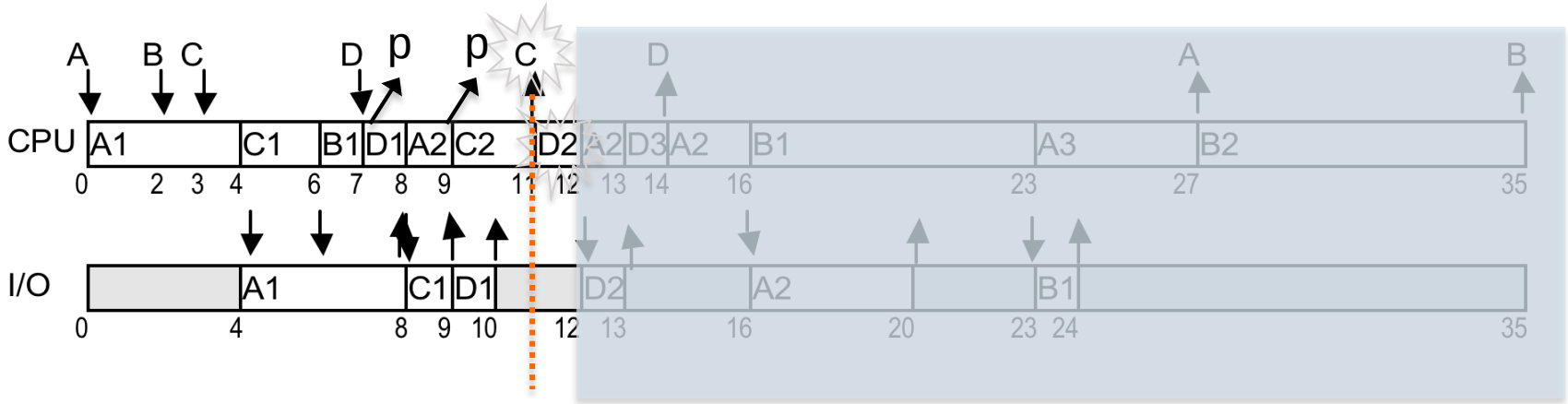


Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 3	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

RQ : B:8:7 | A:4:3 | ~~D:1:1~~

Proc in CPU:



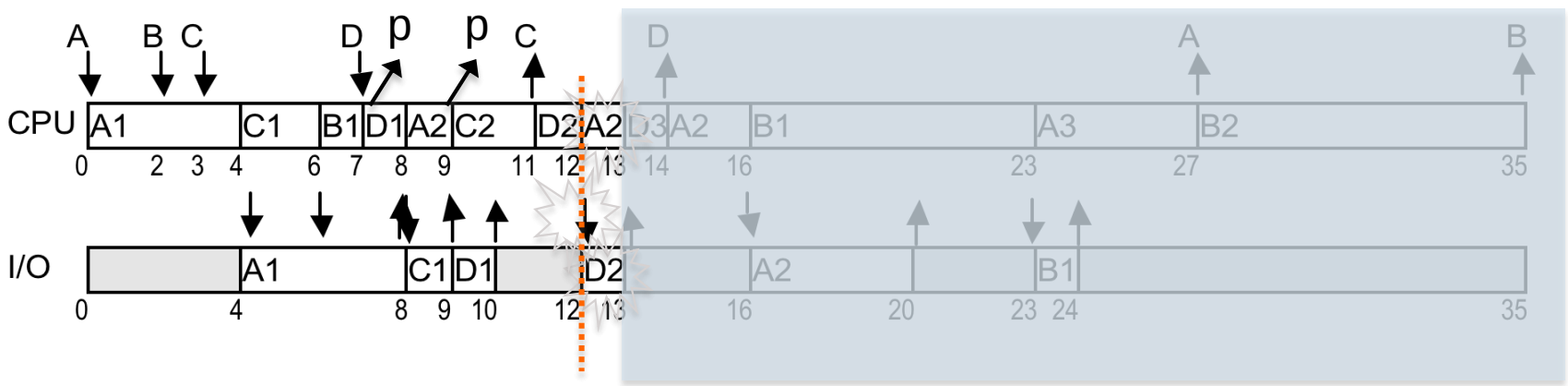
Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 3	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 1

RQ :

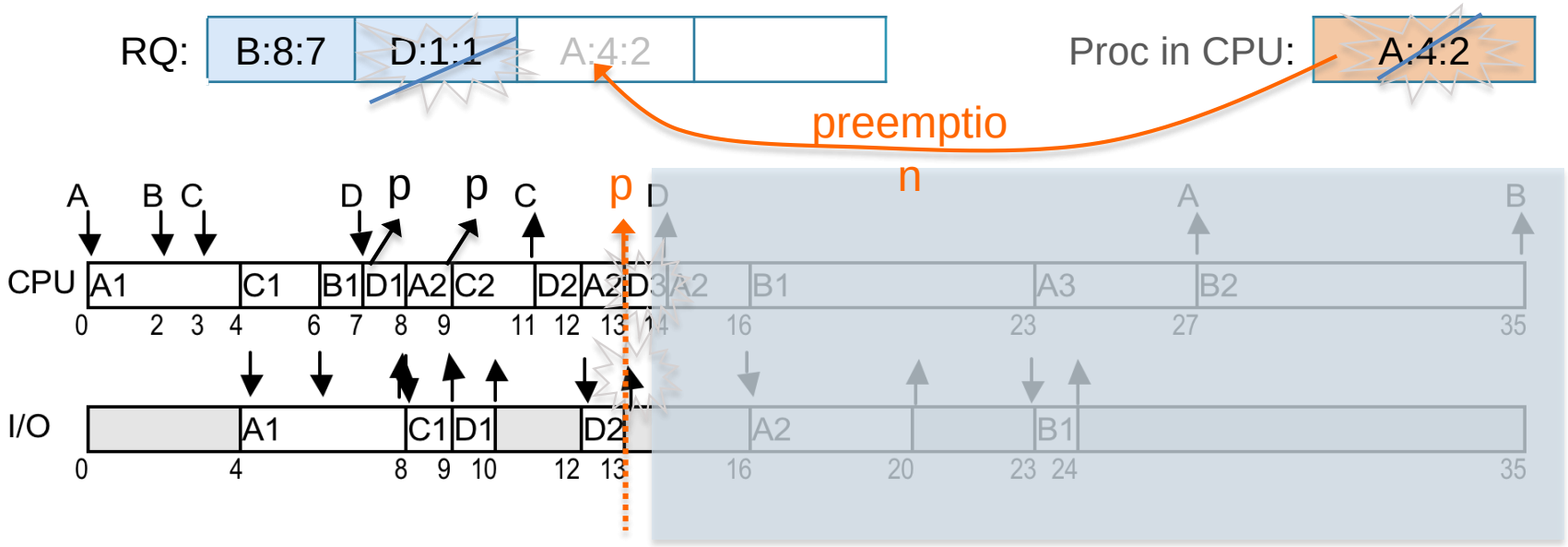
B:8:7	~~A:4.3~~		

Proc in CPU:



Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 2	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 1



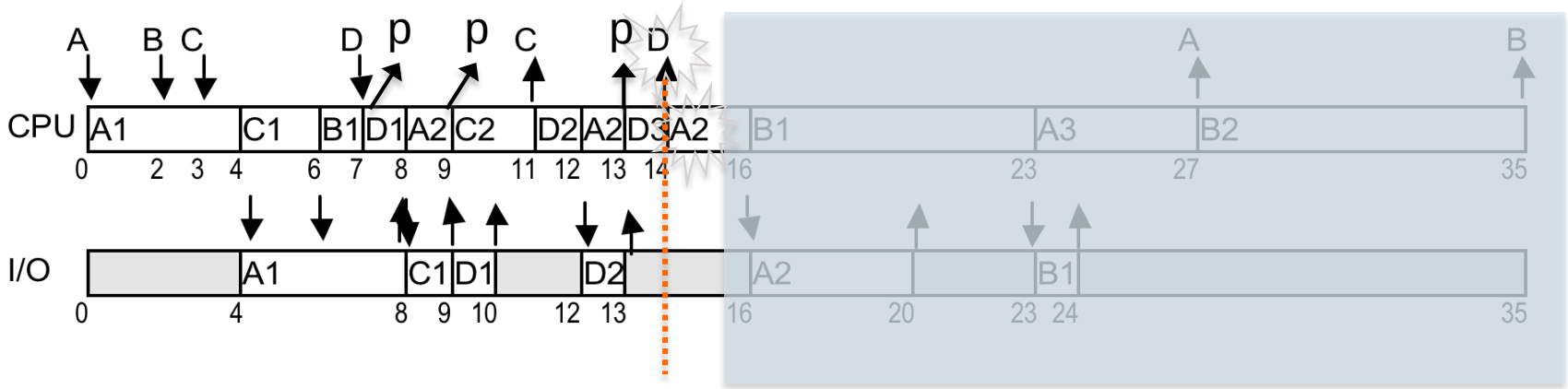
Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 2	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

RQ :

B:8:7	~~A:4:2~~		

Proc in CPU:

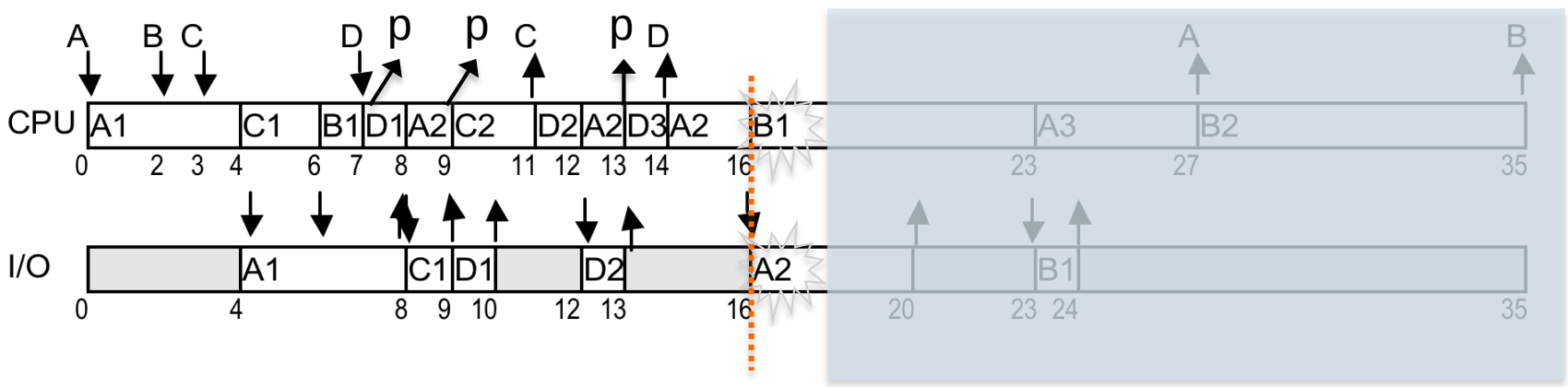


Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 4
B	2	8 : 7	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

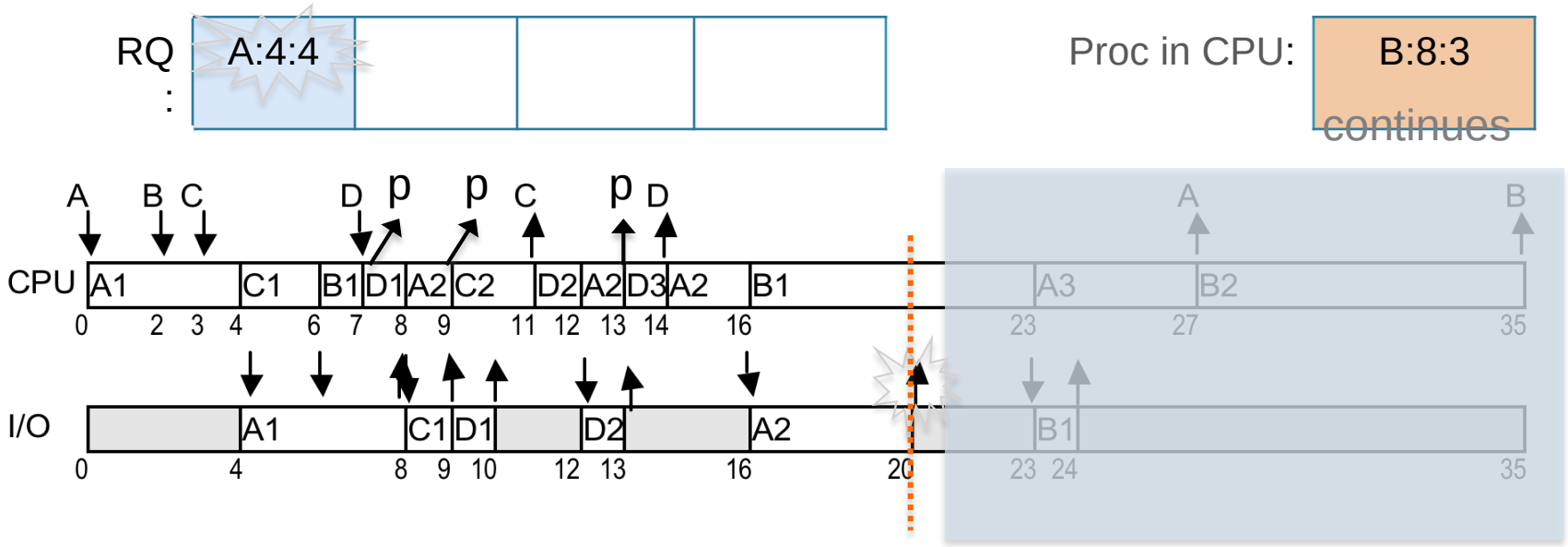
RQ : ~~B:8:7~~

Proc in CPU:



Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 4
B	2	8 : 3	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

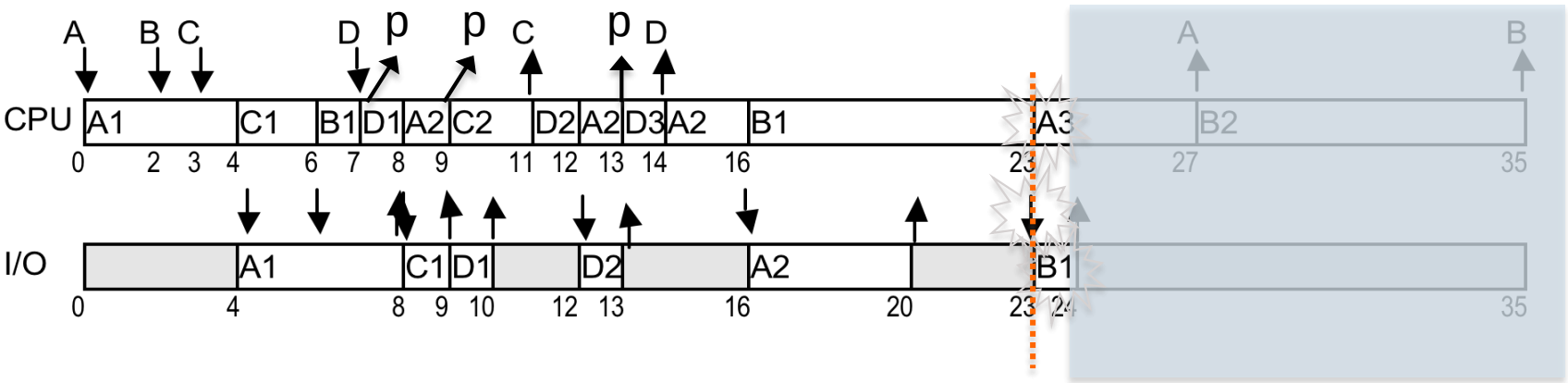


Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 4
B	2	8 : 0	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

RQ :

Proc in CPU:

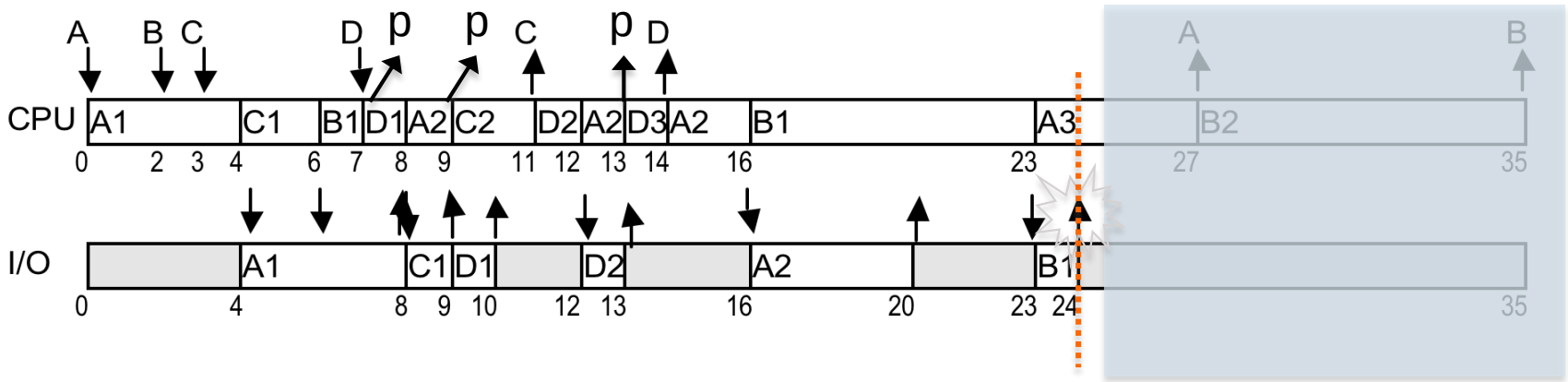


Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 3
B	2	8 : 0	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

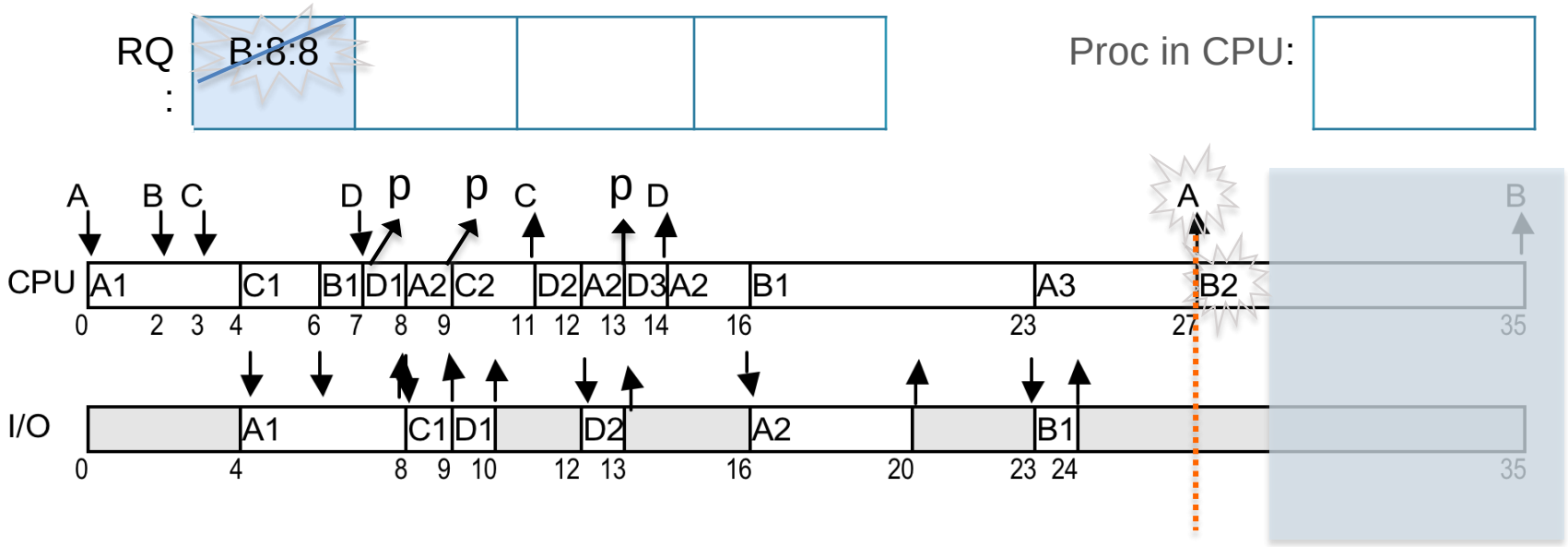
RQ : B:8:8

Proc in CPU: A:4:3 continues



Shortest-Remaining-Time-First (SRTF)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 0
B	2	8 : 0	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

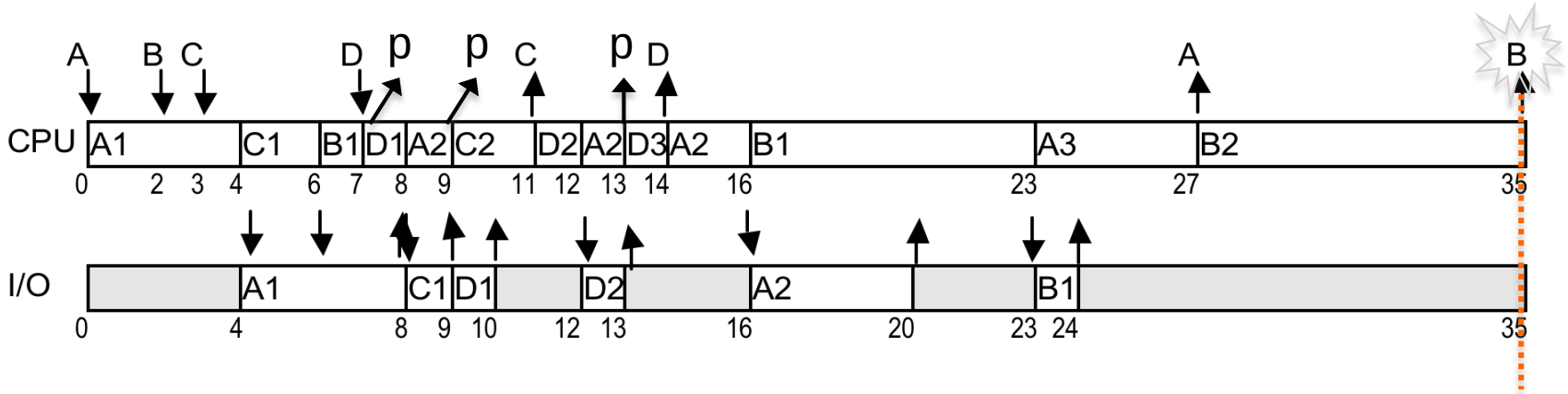


Shortest-Remaining-Time-First (SRTF)

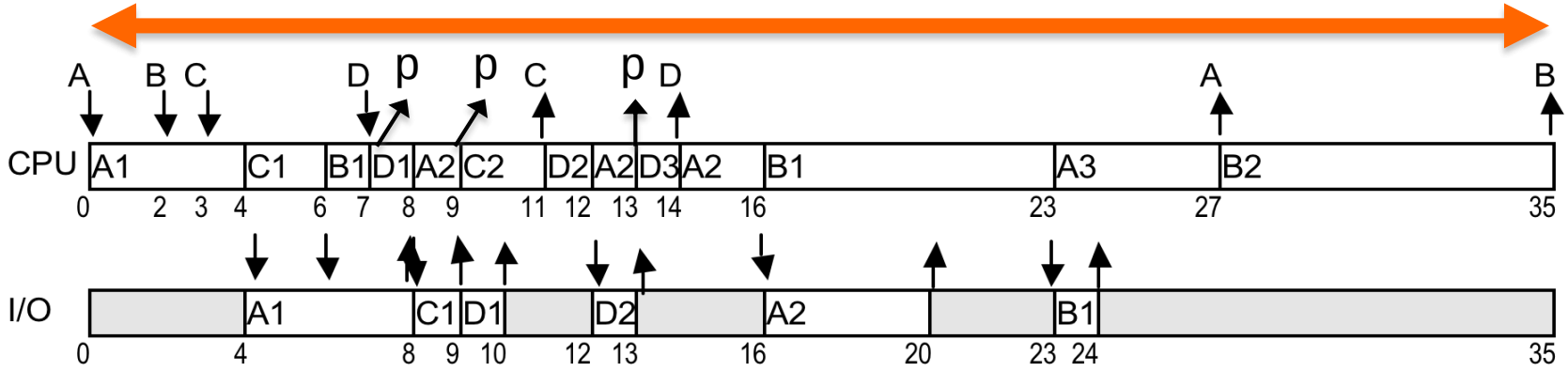
Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 0
B	2	8 : 0	1	8 : 0	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

RQ :

Proc in CPU:

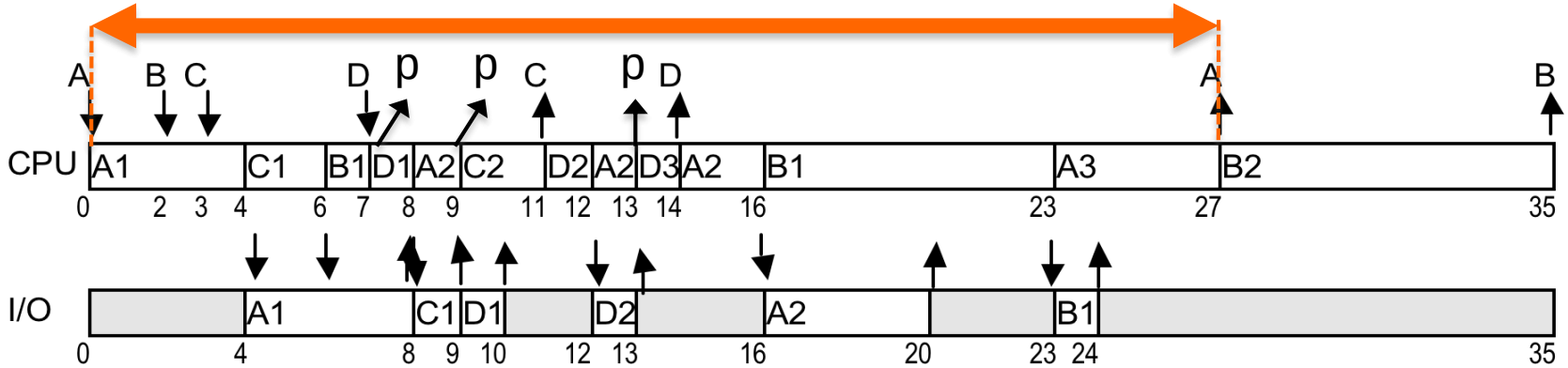


Shortest-Remaining-Time-First (SRTF)



- Processor utilization = (35 / 35) * 100 = 100 %
- Throughput = 4 / 35 = 0.11

Shortest-Remaining-Time-First (SRTF)



- Turn around time:

  $tat_A = 27 - 0 = 27$

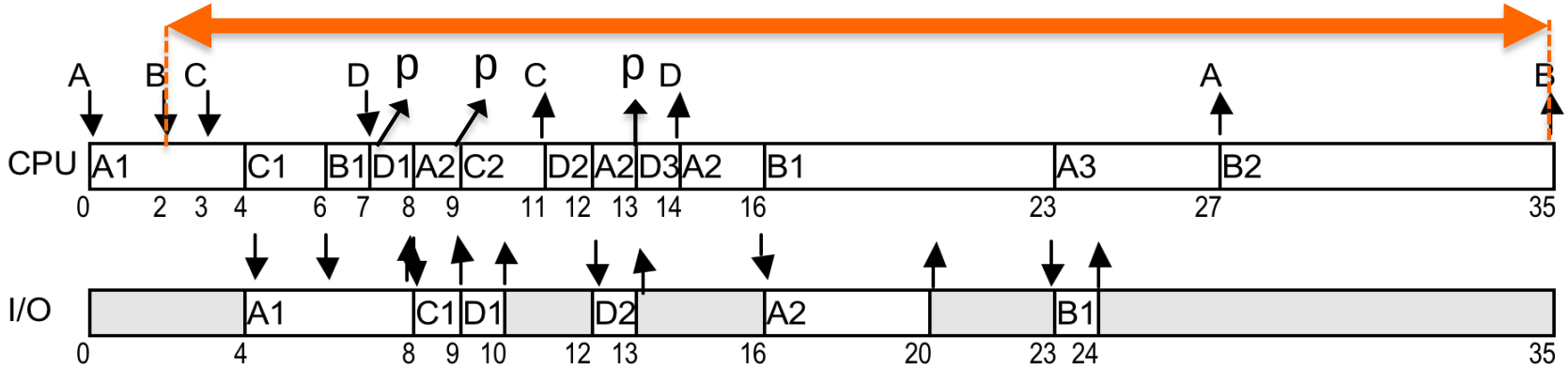
  $tat_B = 35 - 2 = 33$

  $tat_C = 11 - 3 = 8$

  $tat_D = 14 - 7 = 7$

  $tat_{AVG} = (27 + 33 + 8 + 7) / 4 = 18.75$

Shortest-Remaining-Time-First (SRTF)



- Turn around time:

  $tat_A = 27 - 0 = 27$

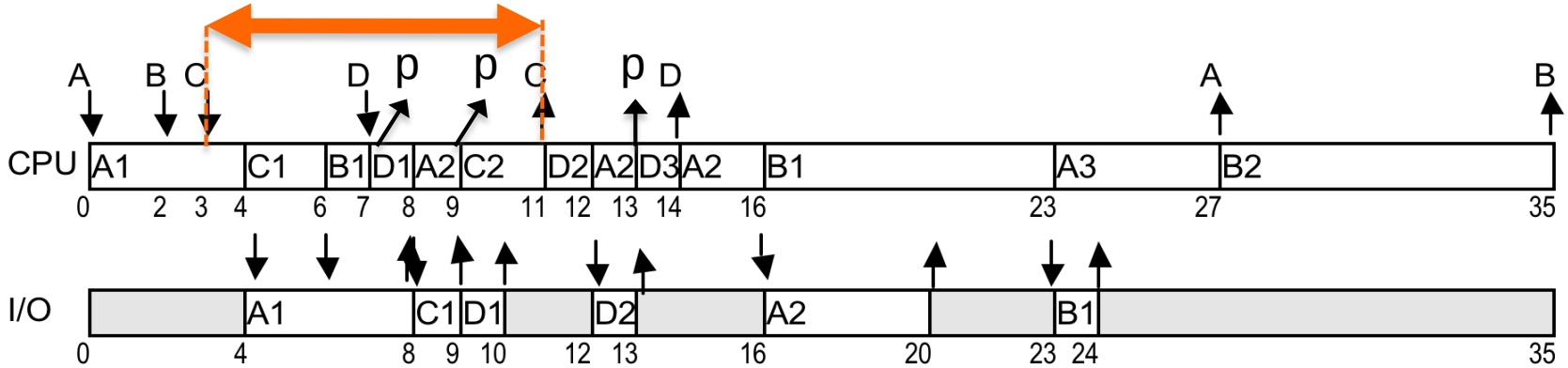
  $tat_B = 35 - 2 = 33$

  $tat_C = 11 - 3 = 8$

  $tat_D = 14 - 7 = 7$

  $tat_{AVG} = (27 + 33 + 8 + 7) / 4 = 18.75$

Shortest-Remaining-Time-First (SRTF)



- Turn around time:

  $tat_A = 27 - 0 = 27$

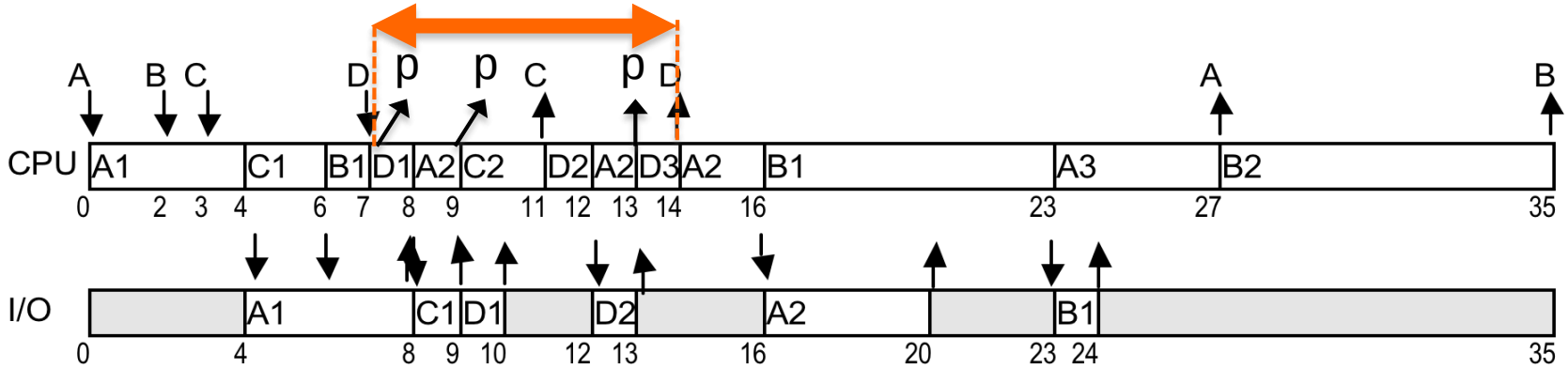
  $tat_B = 35 - 2 = 33$

  $tat_C = 11 - 3 = 8$

  $tat_D = 14 - 7 = 7$

  $tat_{AVG} = (27 + 33 + 8 + 7) / 4 = 18.75$

Shortest-Remaining-Time-First (SRTF)



- Turn around time:

  $tat_A = 27 - 0 = 27$

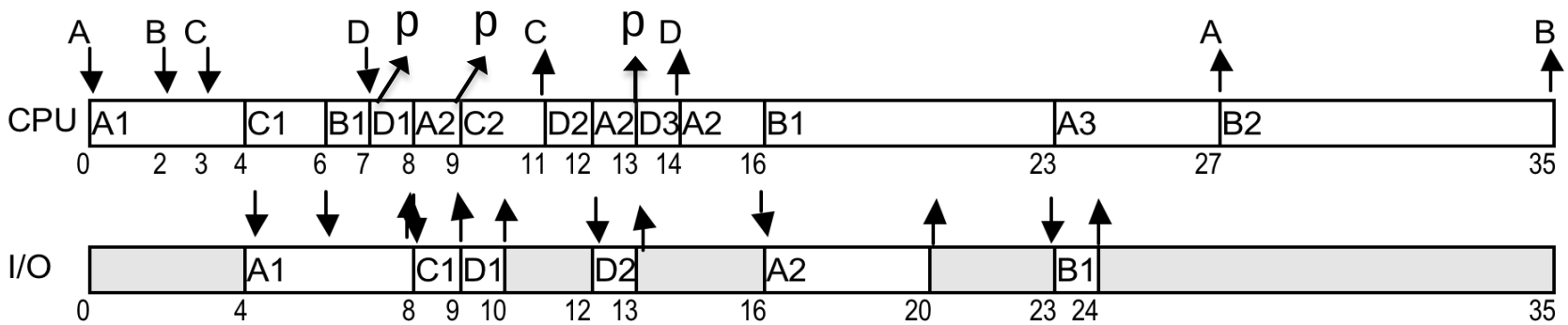
  $tat_B = 35 - 2 = 33$

  $tat_C = 11 - 3 = 8$

  $tat_D = 14 - 7 = 7$

  $tat_{AVG} = (27 + 33 + 8 + 7) / 4 = 18.75$

Shortest-Remaining-Time-First (SRTF)



- Turn around time:

  $tat_A = 27 - 0 = 27$

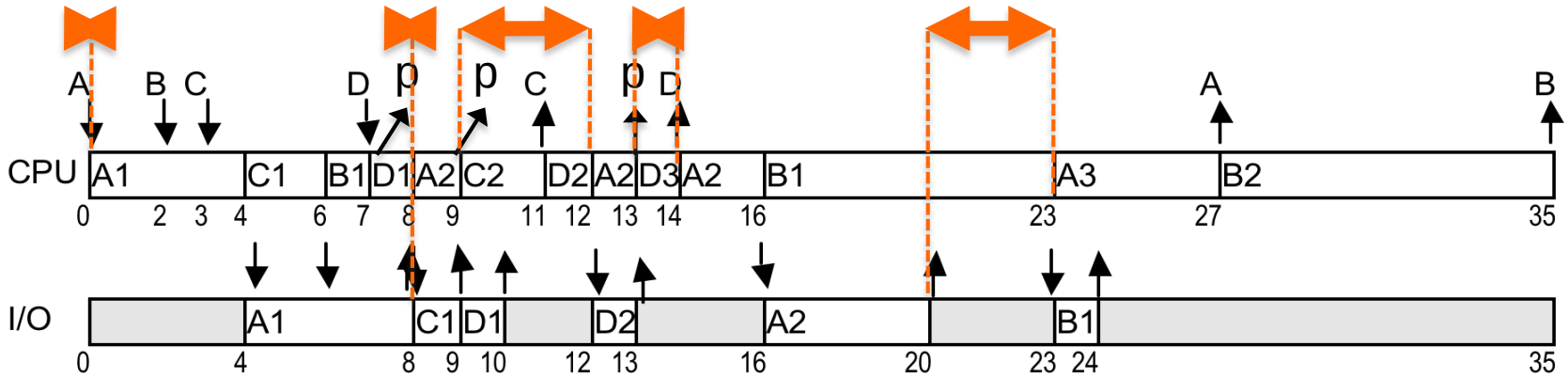
  $tat_B = 35 - 2 = 33$

  $tat_C = 11 - 3 = 8$

  $tat_D = 14 - 7 = 7$

  $tat_{AVG} = (27 + 33 + 8 + 7) / 4 = 18.75$

Shortest-Remaining-Time-First (SRTF)



- Waiting time:

  $wt_A = (0 – 0) + (8 – 8) + (12 - 9) + (14 – 13) + (23 - 20) = 7$

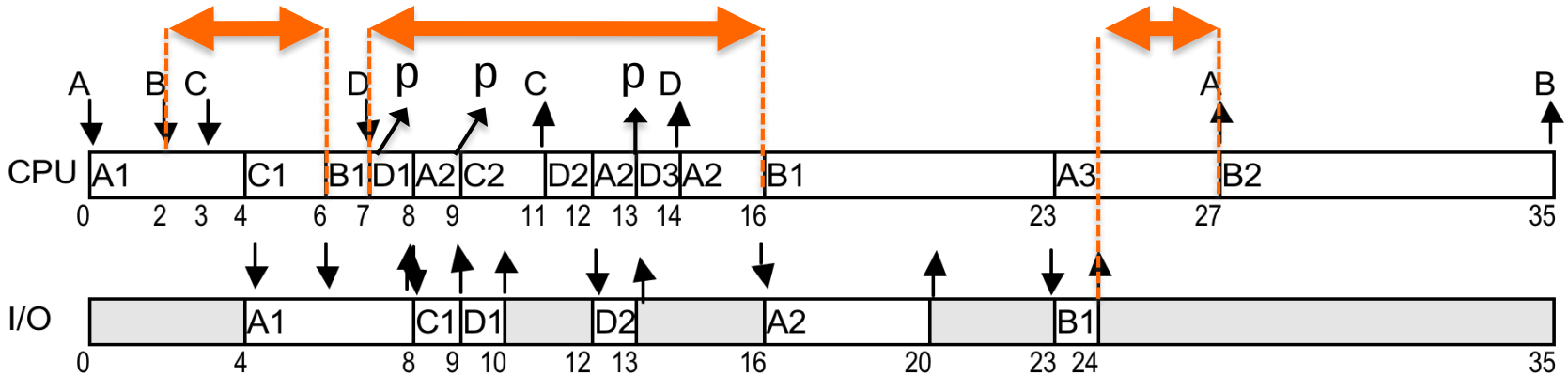
  $wt_B = (6 – 2) + (16 – 7) + (27-24) = 16$

  $wt_C = (4 – 3) + (9 – 9) = 1$

  $wt_D = (7 – 7) + (11 – 10) + (13 – 13) = 1$

  $wt_{AVG} = (7 + 16 + 1 + 1) / 4 = 6.25$

Shortest-Remaining-Time-First (SRTF)



- Waiting time:

  $wt_A = (0 – 0) + (8 – 8) + (12 - 9) + (14 – 13) + (23 - 20) = 7$

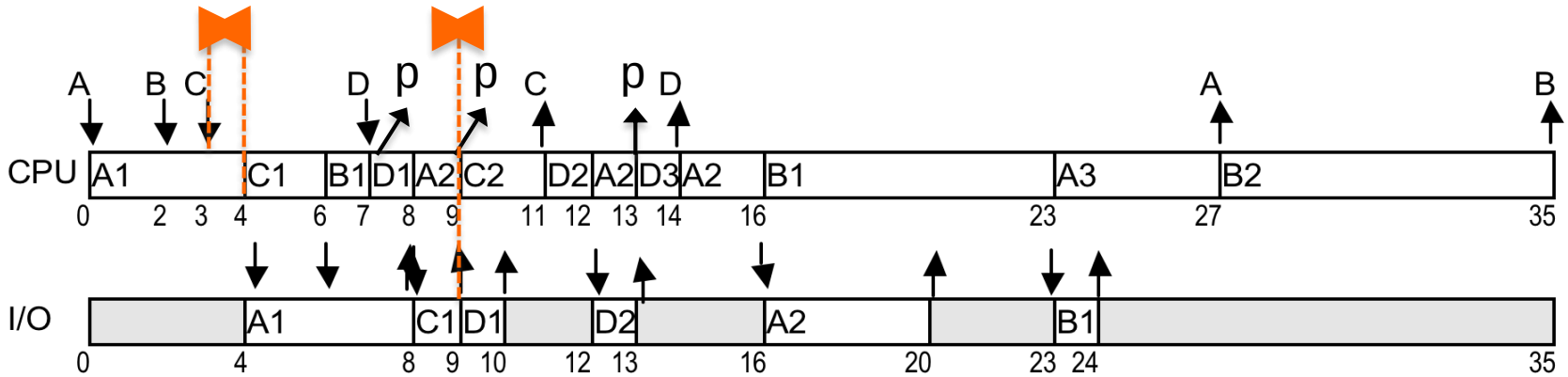
  $wt_B = (6 – 2) + (16 – 7) + (27-24) = 16$

  $wt_C = (4 – 3) + (9 – 9) = 1$

  $wt_D = (7 – 7) + (11 – 10) + (13 – 13) = 1$

  $wt_{AVG} = (7 + 16 + 1 + 1) / 4 = 6.25$

Shortest-Remaining-Time-First (SRTF)



- Waiting time:

  $wt_A = (0 – 0) + (8 – 8) + (12 - 9) + (14 – 13) + (23 - 20) = 7$

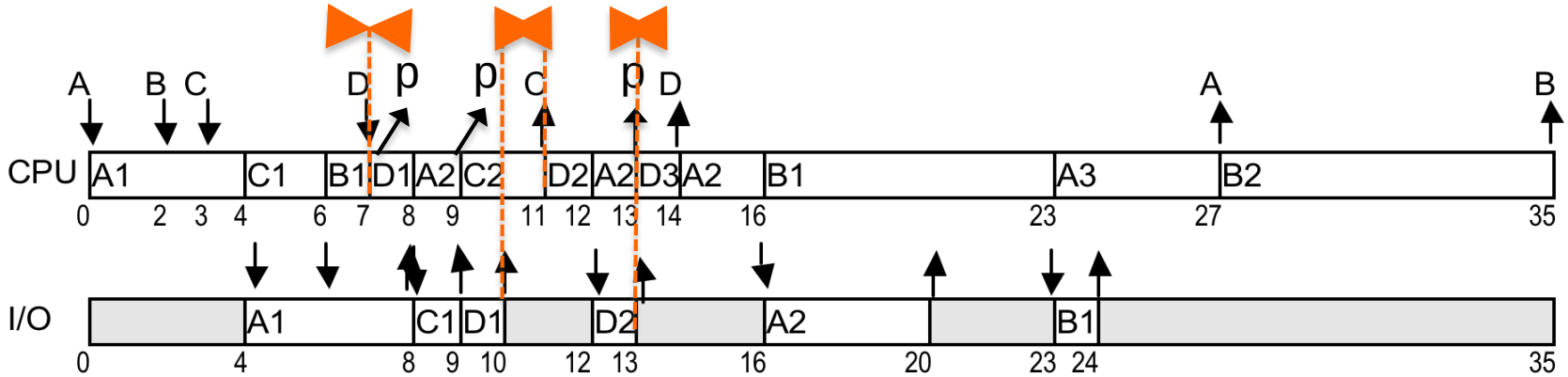
  $wt_B = (6 – 2) + (16 – 7) + (27-24) = 16$

  $wt_C = (4 – 3) + (9 – 9) = 1$

  $wt_D = (7 – 7) + (11 – 10) + (13 – 13) = 1$

  $wt_{AVG} = (7 + 16 + 1 + 1) / 4 = 6.25$

Shortest-Remaining-Time-First (SRTF)



- Waiting time:

  $wt_A = (0 – 0) + (8 – 8) + (12 - 9) + (14 – 13) + (23 - 20) = 7$

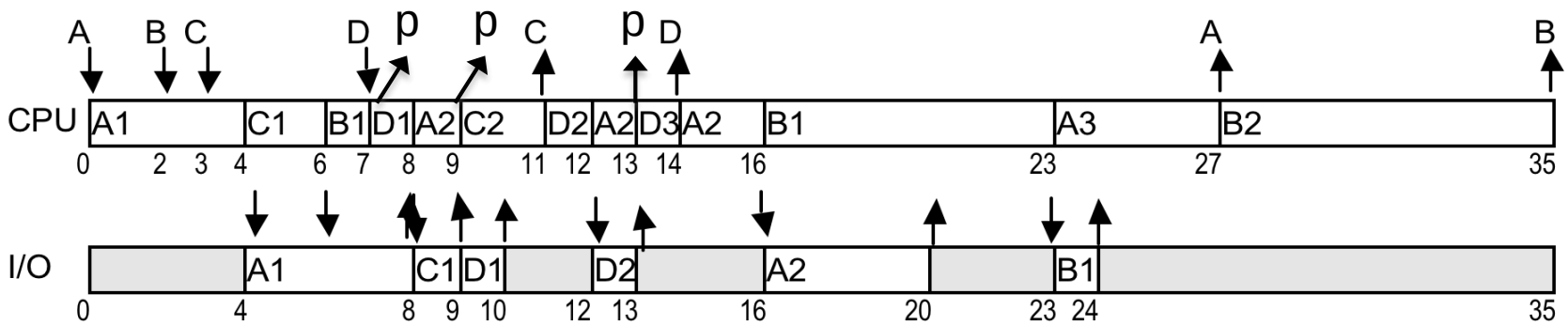
  $wt_B = (6 – 2) + (16 – 7) + (27-24) = 16$

  $wt_C = (4 – 3) + (9 – 9) = 1$

  $wt_D = (7 – 7) + (11 – 10) + (13 – 13) = 1$

  $wt_{AVG} = (7 + 16 + 1 + 1) / 4 = 6.25$

Shortest-Remaining-Time-First (SRTF)



- Waiting time:

$wt_A = (0 – 0) + (8 – 8) + (12 - 9) + (14 – 13) + (23 - 20) = 7$

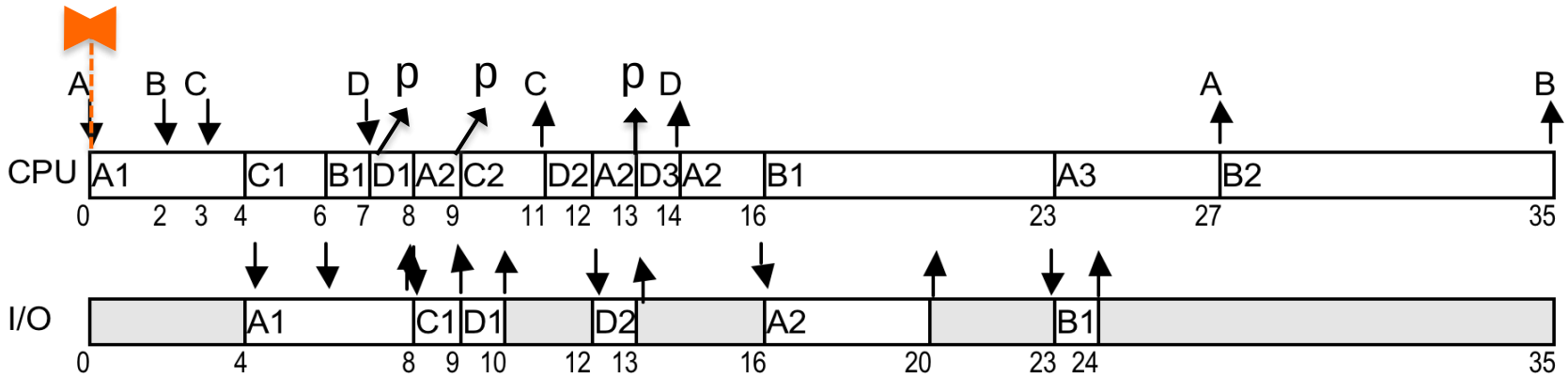
$wt_B = (6 – 2) + (16 – 7) + (27-24) = 16$

$wt_C = (4 – 3) + (9 – 9) = 1$

$wt_D = (7 – 7) + (11 – 10) + (13 – 13) = 1$

$wt_{AVG} = (7 + 16 + 1 + 1) / 4 = 6.25$

Shortest-Remaining-Time-First (SRTF)



- Response time:

  $rt_A = 0 - 0 = 0$

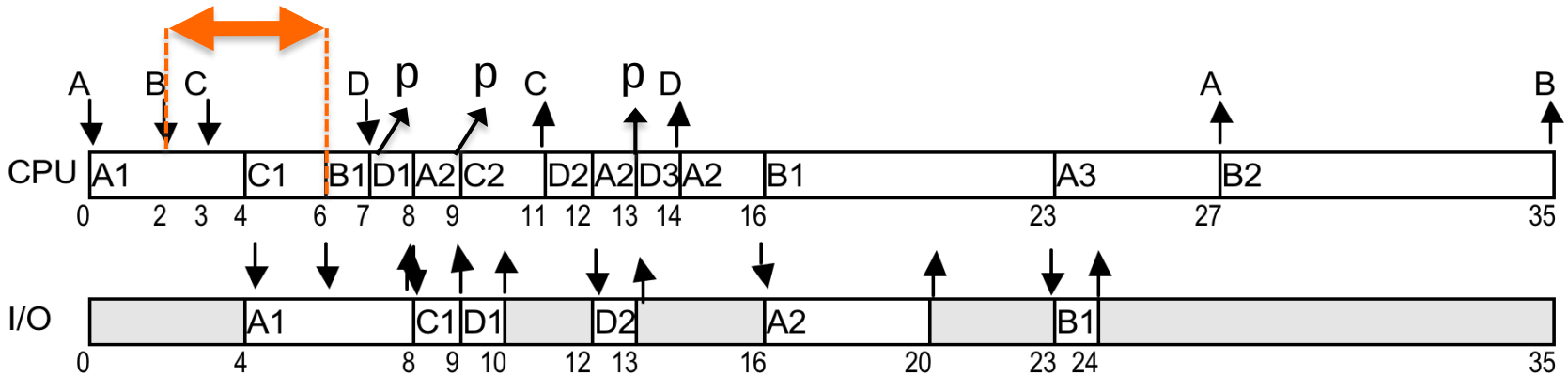
  $rt_B = 6 - 2 = 4$

  $rt_C = 4 - 3 = 1$

  $rt_D = 7 - 7 = 0$

  $rt_{AVG} = (0 + 4 + 1 + 0) / 4 = 1.25$

Shortest-Remaining-Time-First (SRTF)



- Response time:

  $rt_A = 0 - 0 = 0$

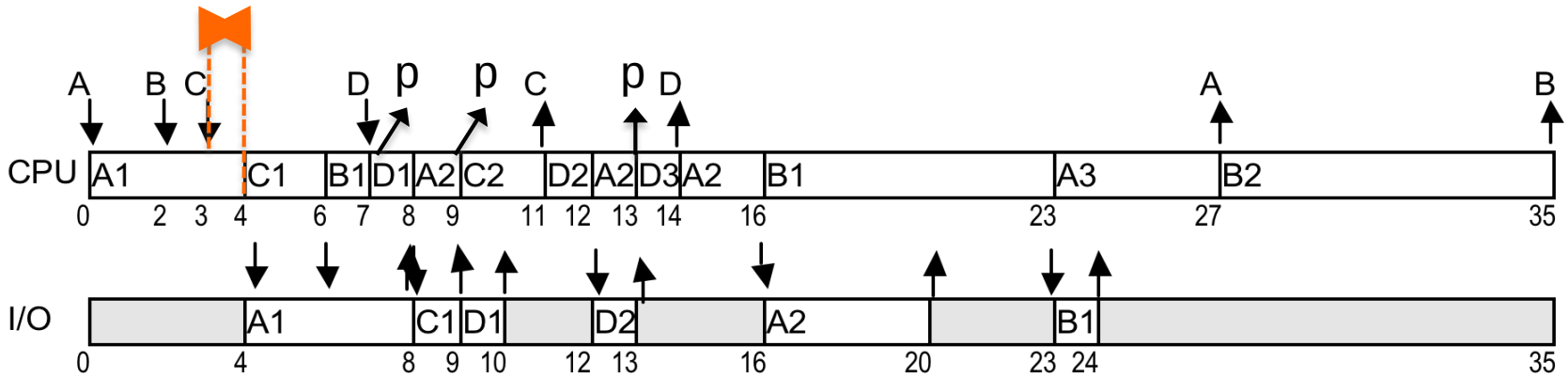
  $rt_B = 6 - 2 = 4$

  $rt_C = 4 - 3 = 1$

  $rt_D = 7 - 7 = 0$

  $rt_{AVG} = (0 + 4 + 1 + 0) / 4 = 1.25$

Shortest-Remaining-Time-First (SRTF)



- Response time:

  $rt_A = 0 - 0 = 0$

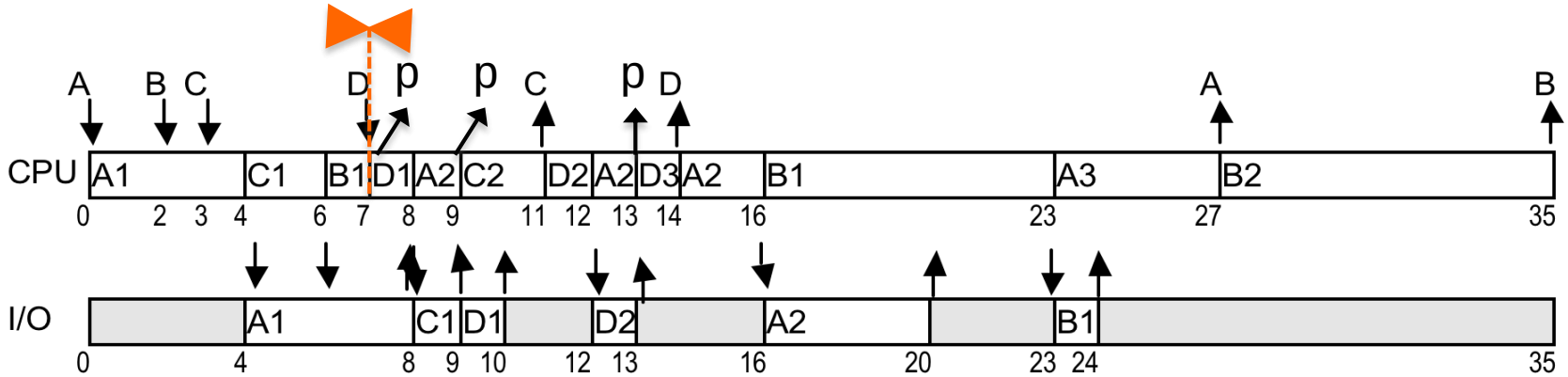
  $rt_B = 6 - 2 = 4$

  $rt_C = 4 - 3 = 1$

  $rt_D = 7 - 7 = 0$

  $rt_{AVG} = (0 + 4 + 1 + 0) / 4 = 1.25$

Shortest-Remaining-Time-First (SRTF)



- Response time:

  $rt_A = 0 - 0 = 0$

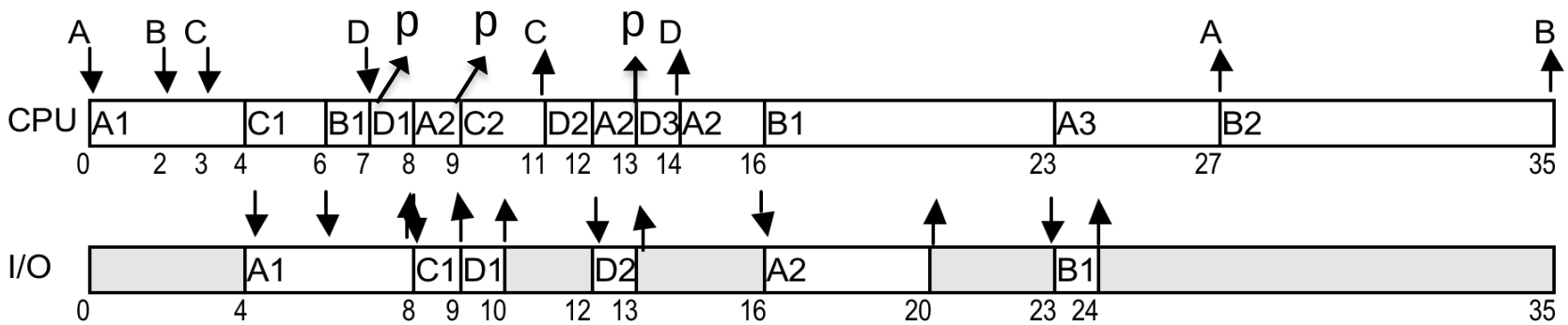
  $rt_B = 6 - 2 = 4$

  $rt_C = 4 - 3 = 1$

  $rt_D = 7 - 7 = 0$

  $rt_{AVG} = (0 + 4 + 1 + 0) / 4 = 1.25$

Shortest-Remaining-Time-First (SRTF)



- Response time:

  $rt_A = 0 - 0 = 0$

  $rt_B = 6 - 2 = 4$

  $rt_C = 4 - 3 = 1$

  $rt_D = 7 - 7 = 0$

  $rt_{AVG} = (0 + 4 + 1 + 0) / 4 = 1.25$

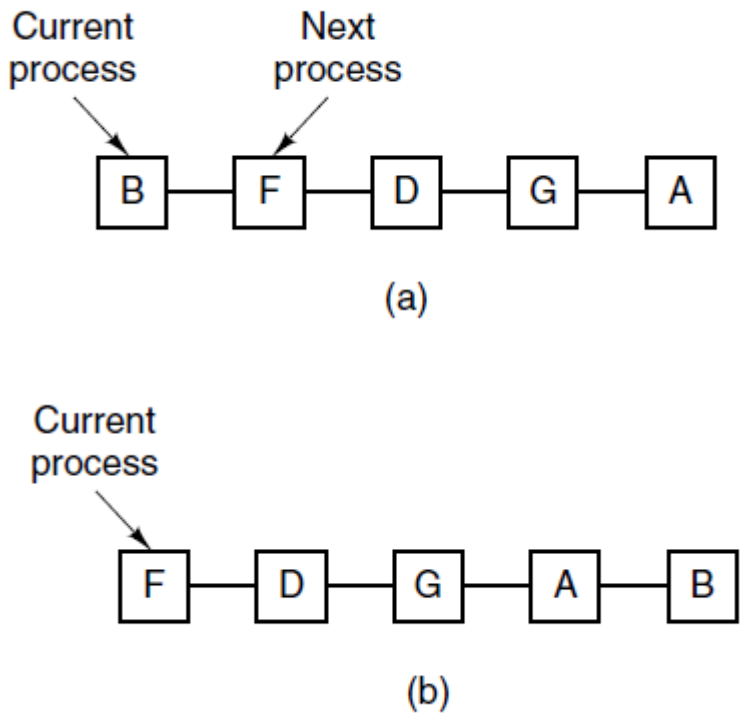
Shortest Job First

An example of shortest job first scheduling.
(a) Running four jobs in the original order.
(b) Running them in shortest job first order.

Scheduling in Interactive Systems

- Round-Robin Scheduling
- Priority Scheduling
- Multiple Queues
- Shortest Process Next
- Guaranteed Scheduling
- Lottery Scheduling
- Fair-Share Scheduling

Round-Robin Scheduling



Round-robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after *B* uses up its quantum.

Round-Robin Scheduling (RRS)

- In RRS algorithm the ready queue is treated as a FIFO circular queue.

- The RRS traces the ready queue allocating the processor to each process for a time interval which is smaller than or equal to a predefined time called **time quantum (slice).**

Round-Robin Scheduling (RRS)

- The OS using RRS, takes the first process from the ready queue, gives the processor to that process and sets a timer to interrupt when time quantum expires.

- If the process has a processor burst time smaller than the time quantum, then it releases the processor voluntarily, either by terminating or by issuing an I/O request.

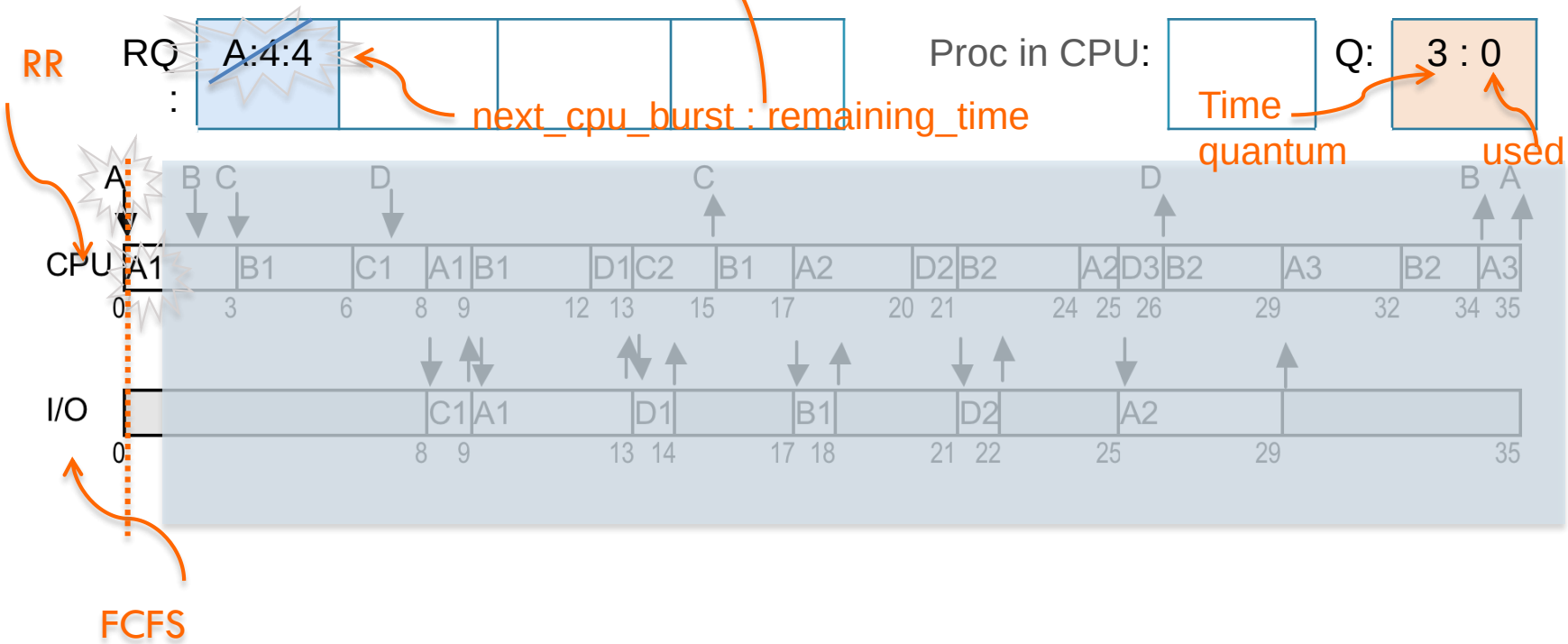
- The OS then proceed with the next process in the ready queue.

Round-Robin Scheduling (RRS)

- The performance of RRS depends heavily on the selected time quantum.
- Time quantum → ∞ $\Rightarrow$ RRS becomes FCFS
- Time quantum → 0 $\Rightarrow$ RRS becomes processor sharing (It acts as if each of the n processes has its own processor running at processor speed divided by n)
- However, if time quantum is too small then context switch time can not be ignored anymore.
- For an optimum time quantum, it can be selected to be greater than 80 % of processor bursts and to be greater than the context switching time.

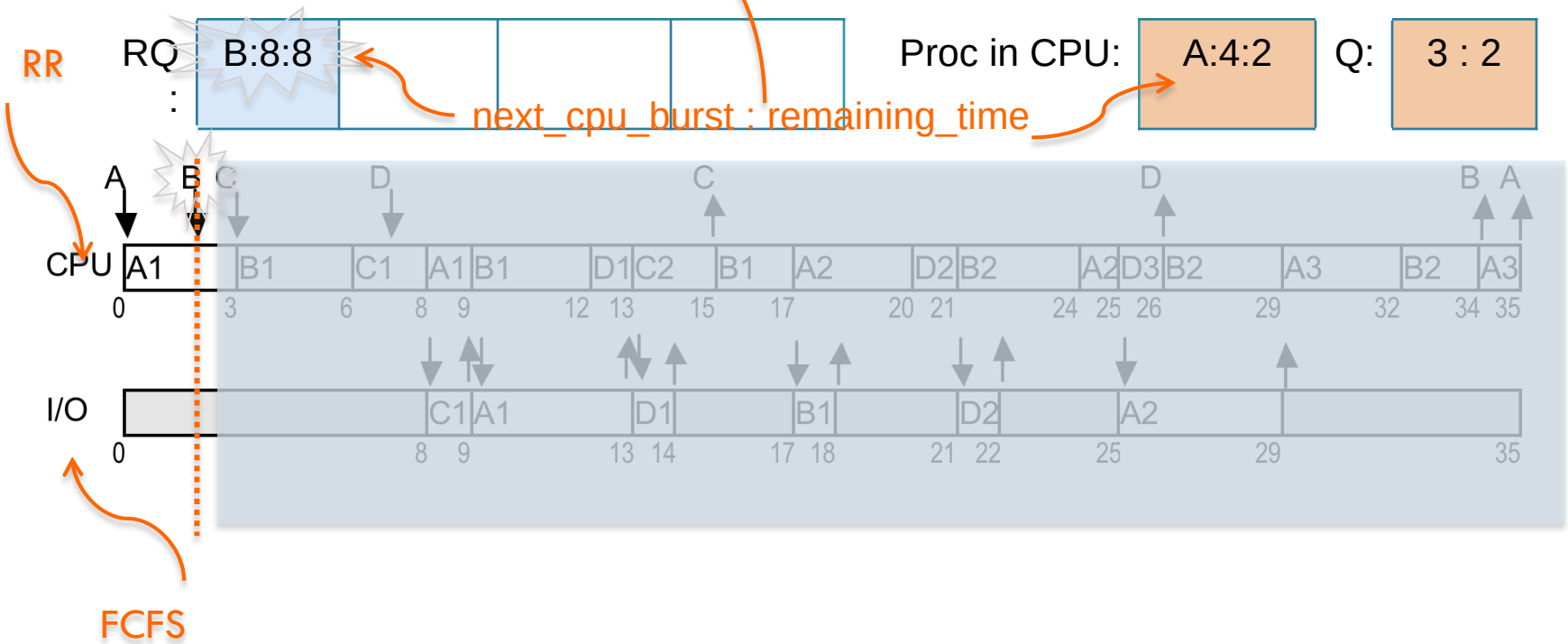
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 4	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1



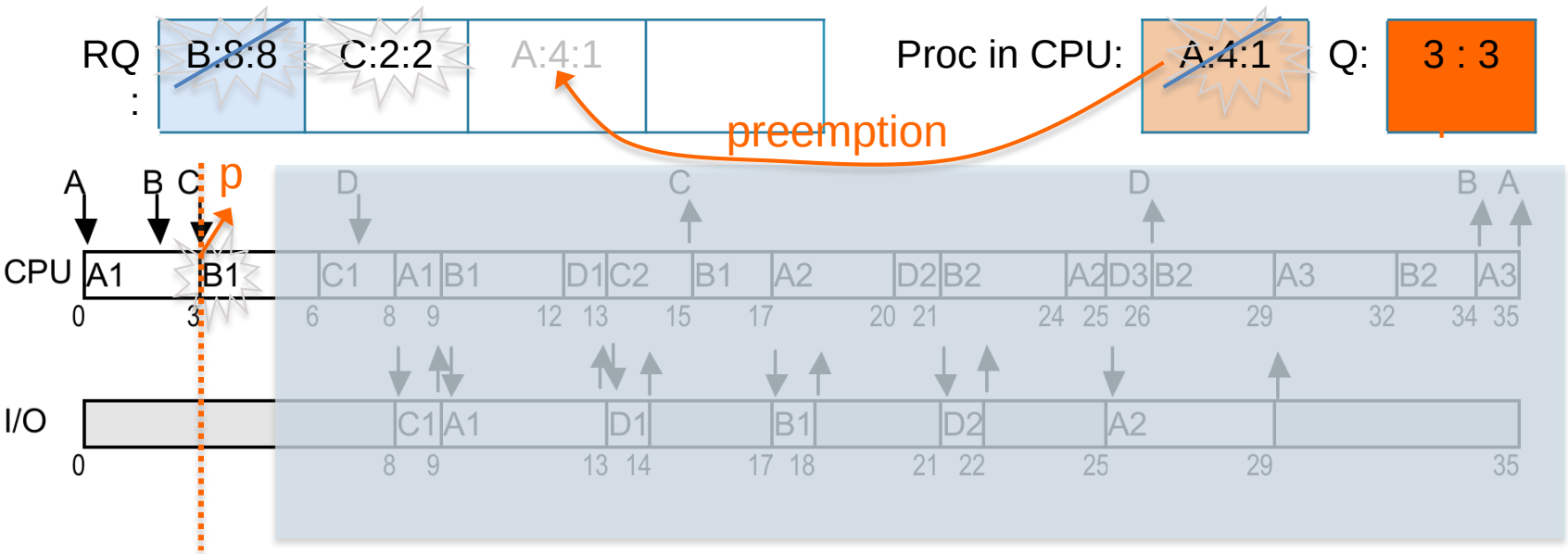
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 2	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

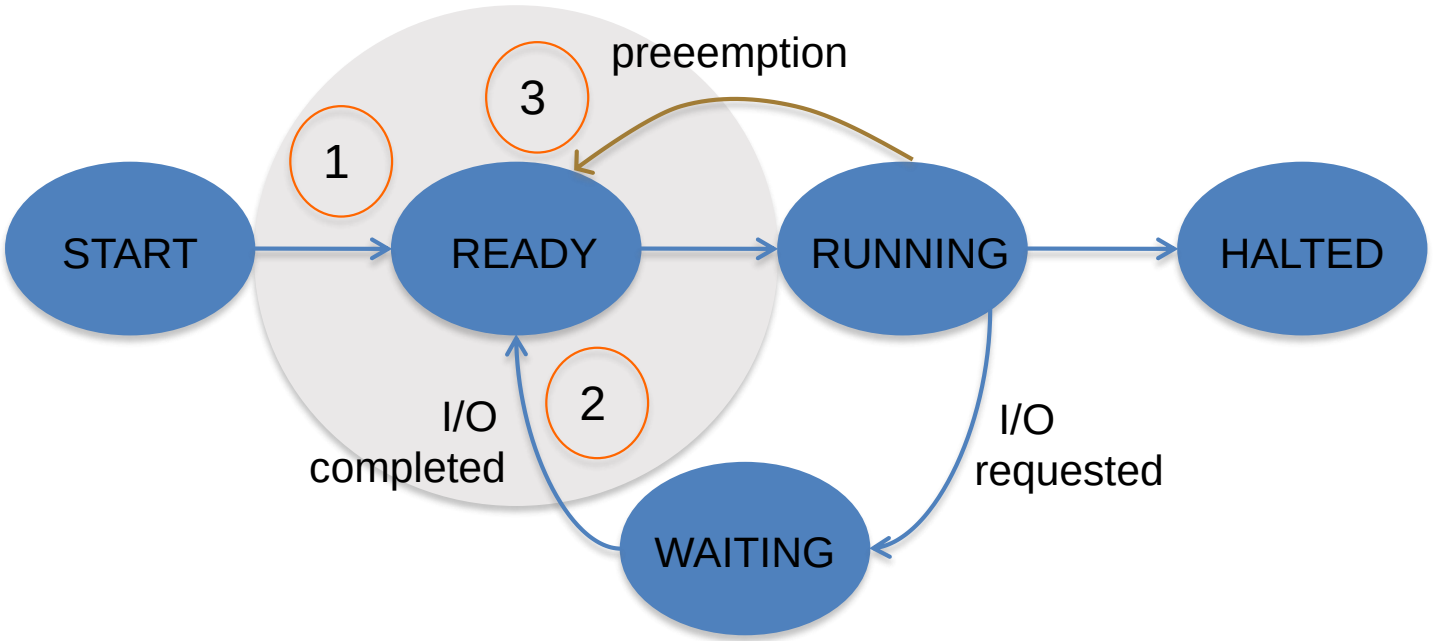


Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 1	4	4 : 4	4	4 : 4
B	2	8 : 8	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1



Process States



If more than one processes are to enter the Ready Queue at the same time then give priority as:

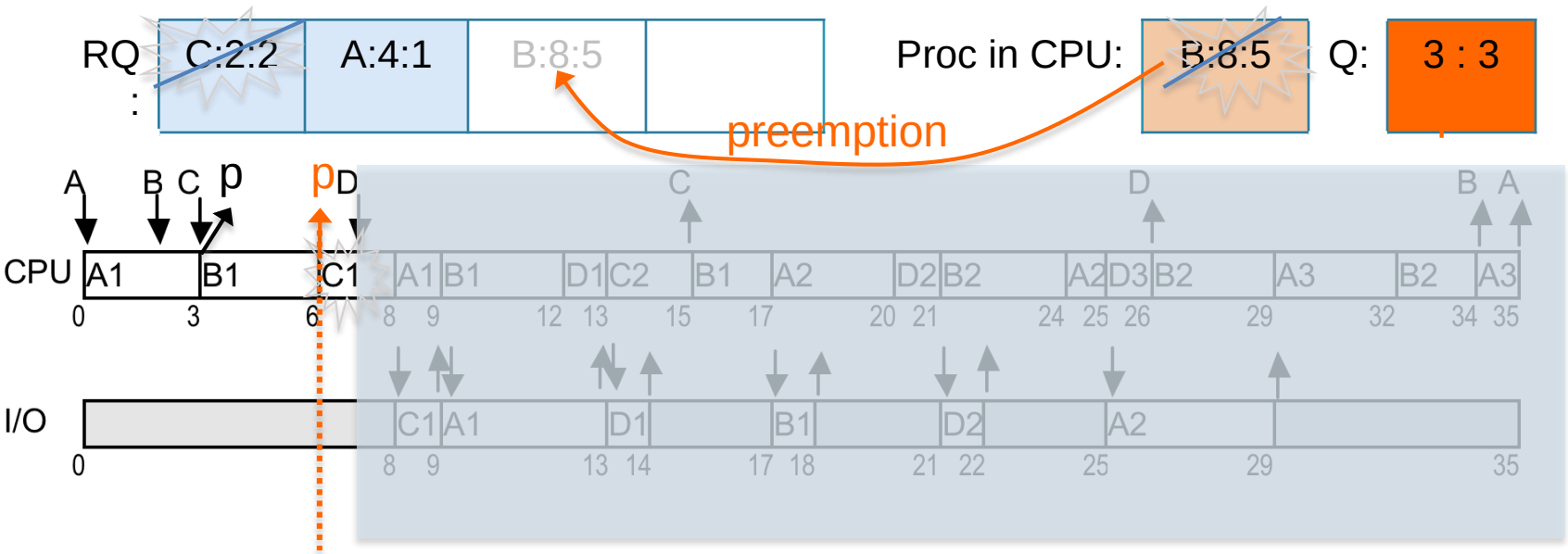
1: process just submitted

2: process completed I/O

3: process preempted CPU

Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 1	4	4 : 4	4	4 : 4
B	2	8 : 5	1	8 : 8	-	-
C	3	2 : 2	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1



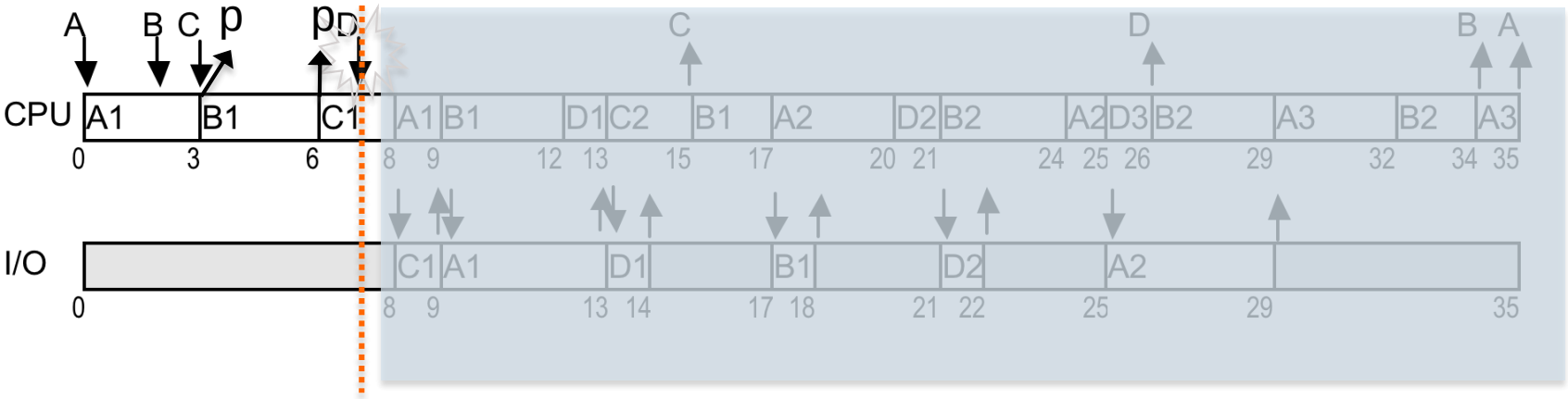
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 1	4	4 : 4	4	4 : 4
B	2	8 : 5	1	8 : 8	-	-
C	3	2 : 1	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

RQ : A:4:1 | B:8:5 | D:1:1

Proc in CPU: C:2:1

Q: 3 : 1 expires



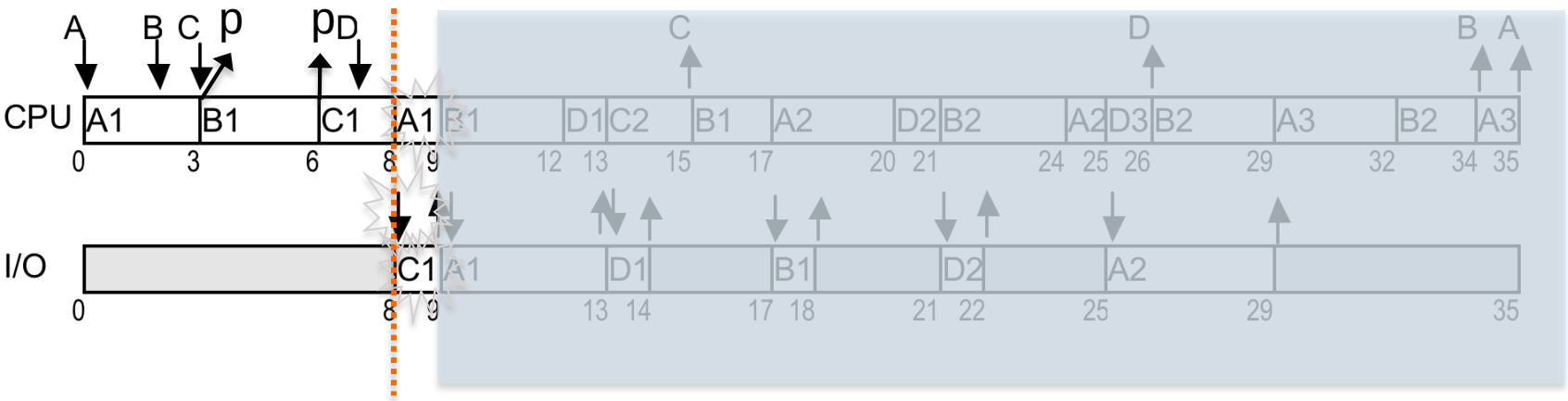
2.3.4 Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 1	4	4 : 4	4	4 : 4
B	2	8 : 5	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

RQ : ~~A:4:1~~ | B:8:5 | D:1:1 |

Proc in CPU:

Q: 3 : 0



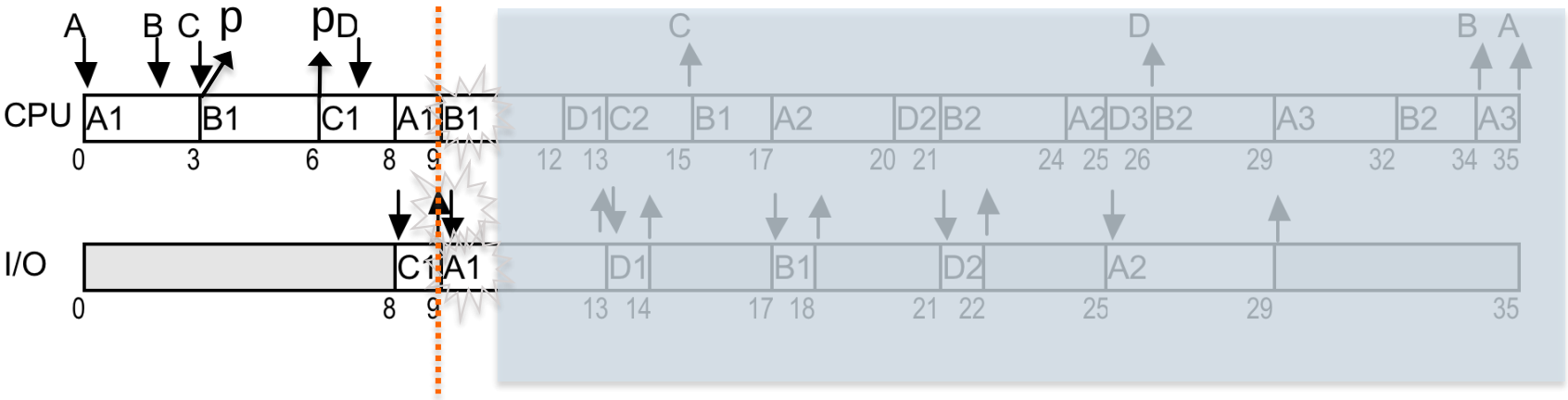
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 5	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1

RQ : ~~B:8:5~~ | D:1:1 | C:2:2

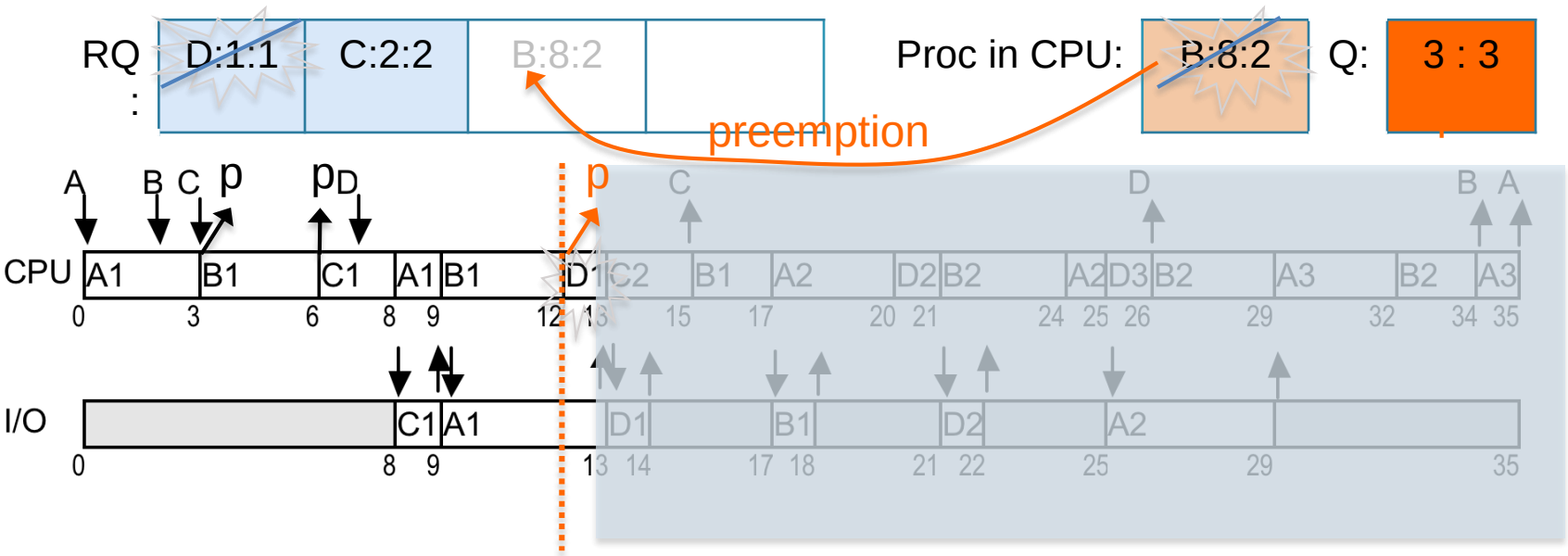
Proc in CPU:

Q: 3 : 0



Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 2	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 1	1	1 : 1	1	1 : 1



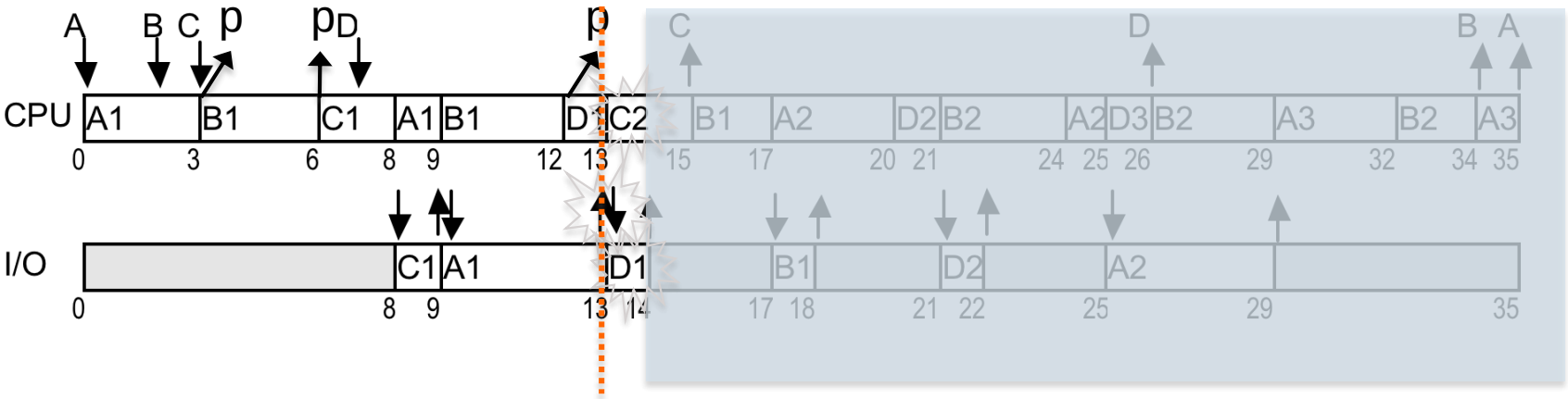
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 2	1	8 : 8	-	-
C	3	2 : 0	1	2 : 2	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

RQ : ~~C:2:2~~ | B:8:2 | A:4:4

Proc in CPU:

Q: 3 : 0



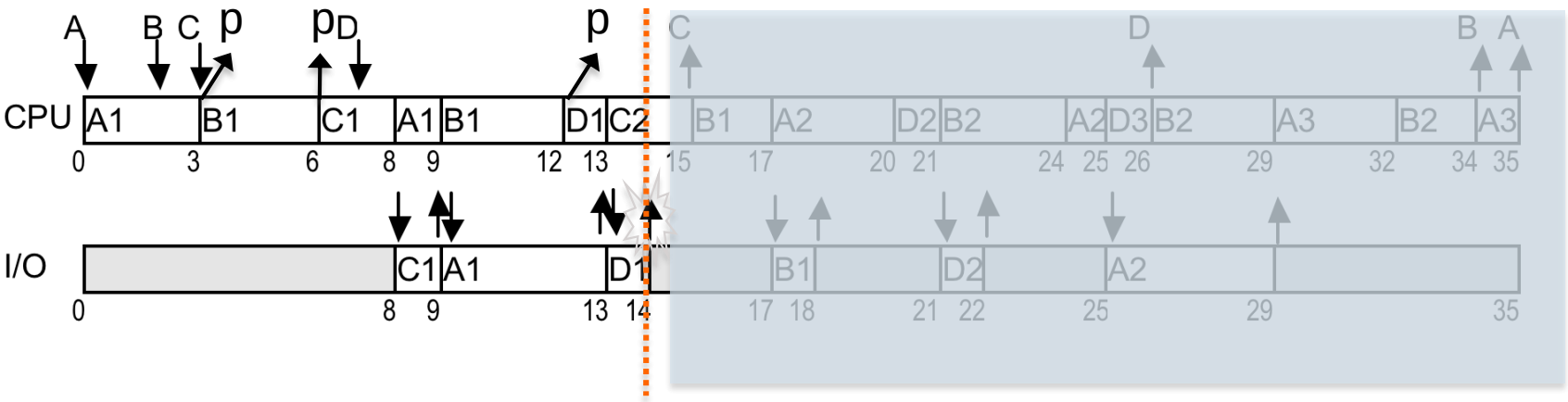
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 2	1	8 : 8	-	-
C	3	2 : 0	1	2 : 1	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

RQ : B:8:2 | A:4:4 | D:1:1

Proc in CPU: C:2:1

Q: 3 : 1 expires



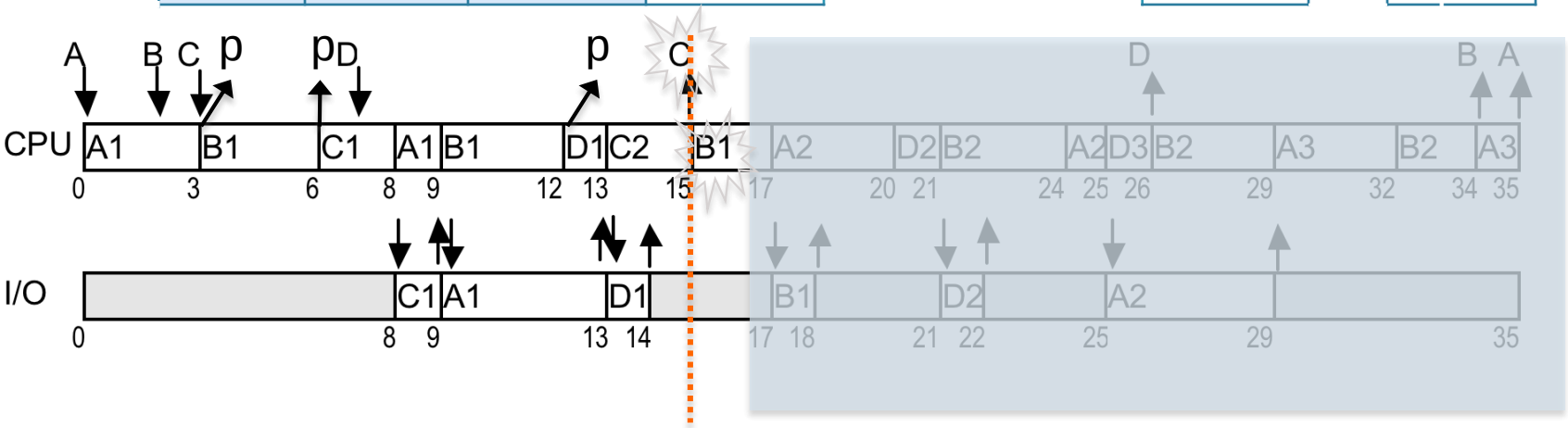
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 2	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

RQ : ~~B:8:2~~ | A:4:4 | D:1:1 |

Proc in CPU:

Q: 3 : 0

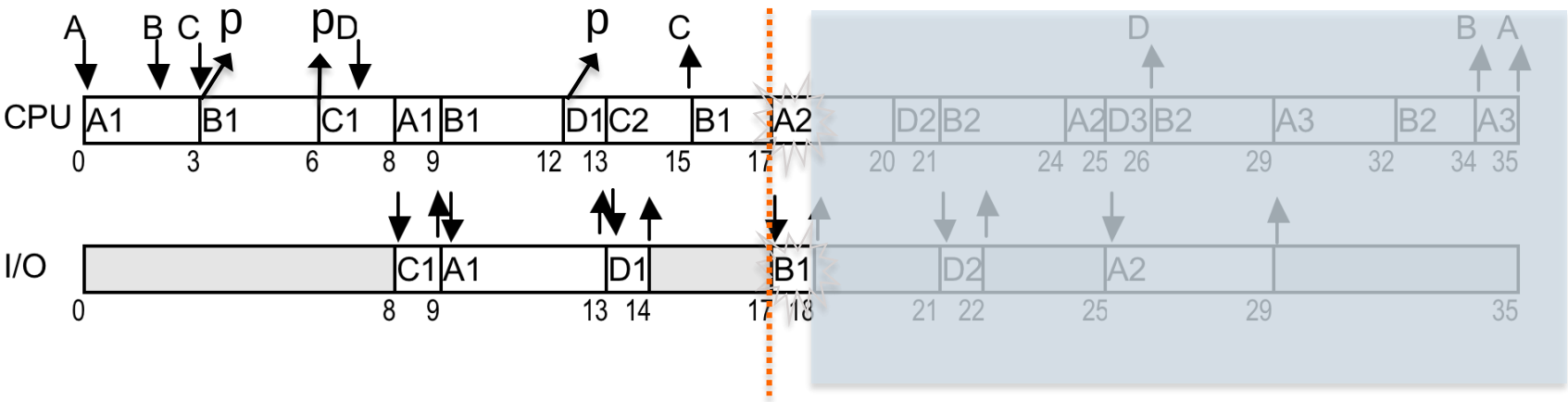


Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 4	4	4 : 4
B	2	8 : 0	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

RQ : ~~A:4:4~~ | D:1:1 | | |

Proc in CPU: | Q: 3 : 0



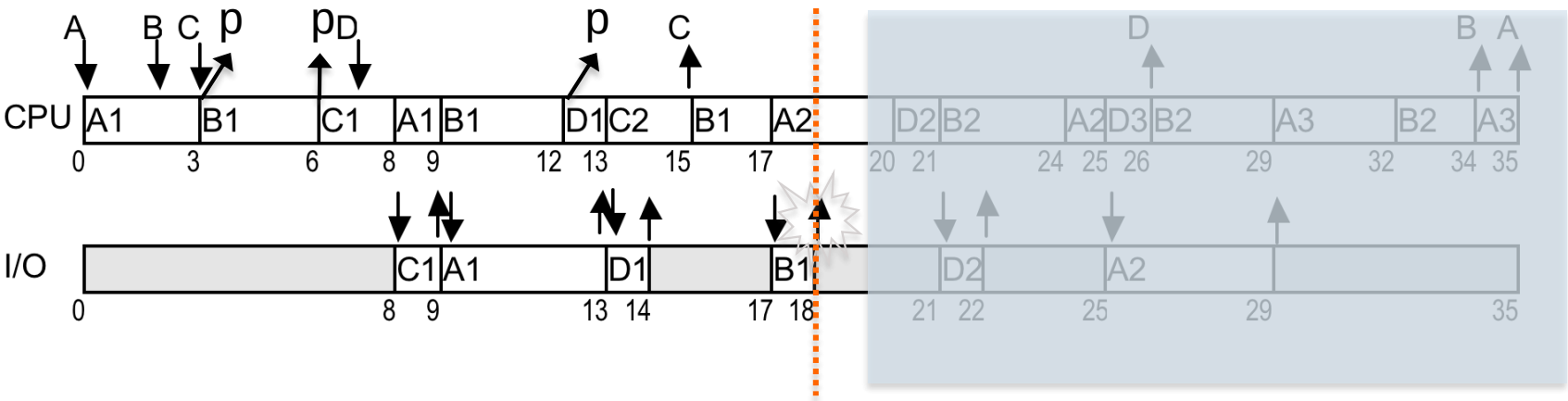
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 3	4	4 : 4
B	2	8 : 0	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

RQ : D:1:1 | B:8:8

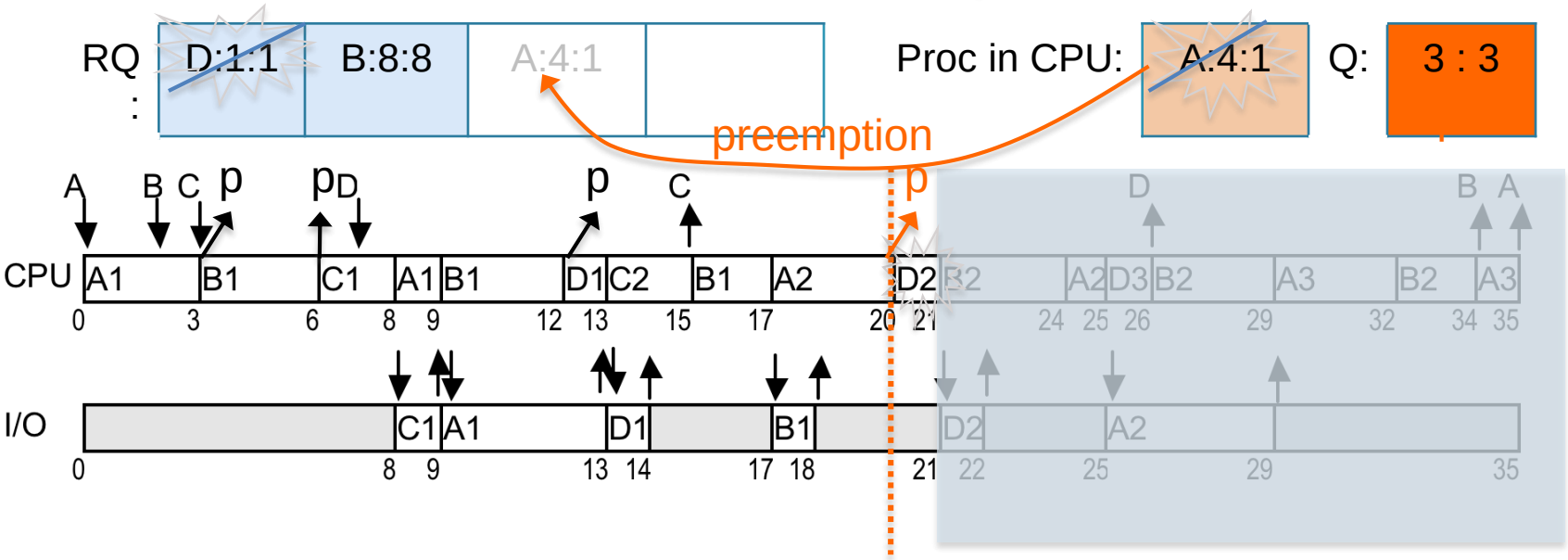
Proc in CPU: A:4:3

Q: 3 : 1 expires



Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 1	4	4 : 4
B	2	8 : 0	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 1	1	1 : 1

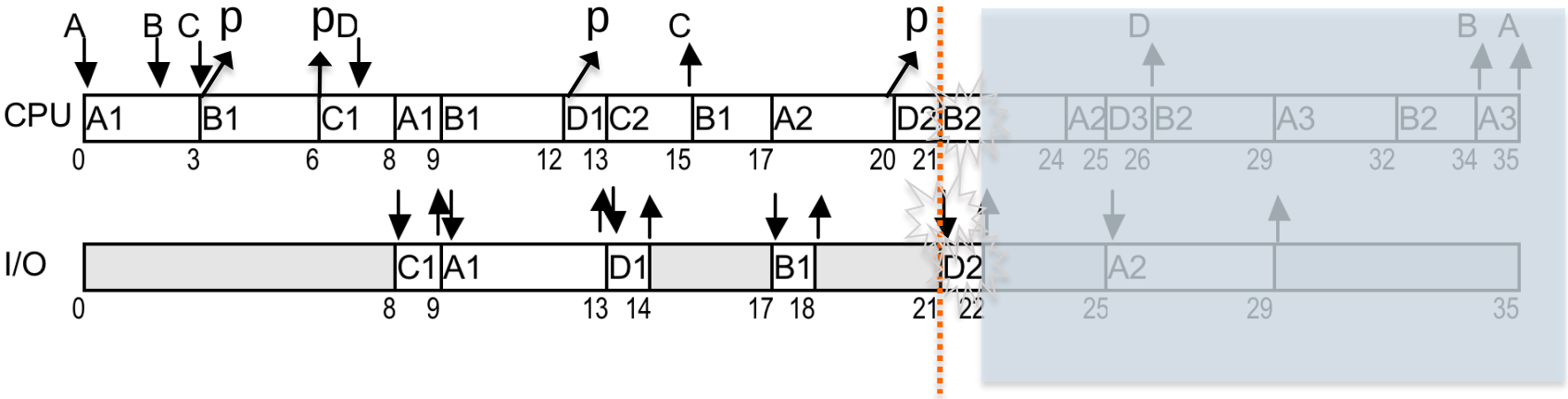


Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 1	4	4 : 4
B	2	8 : 0	1	8 : 8	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 1

RQ : ~~B:8:8~~ | A:4:1 | | |

Proc in CPU: | | Q: 3 : 0



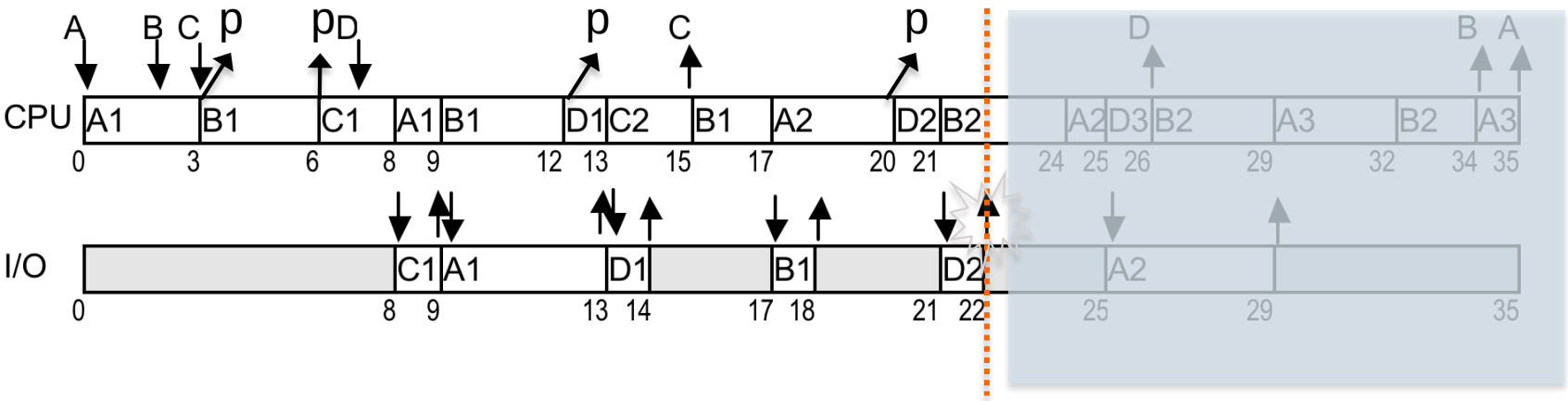
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 1	4	4 : 4
B	2	8 : 0	1	8 : 7	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 1

RQ : A:4:1 | D:1:1

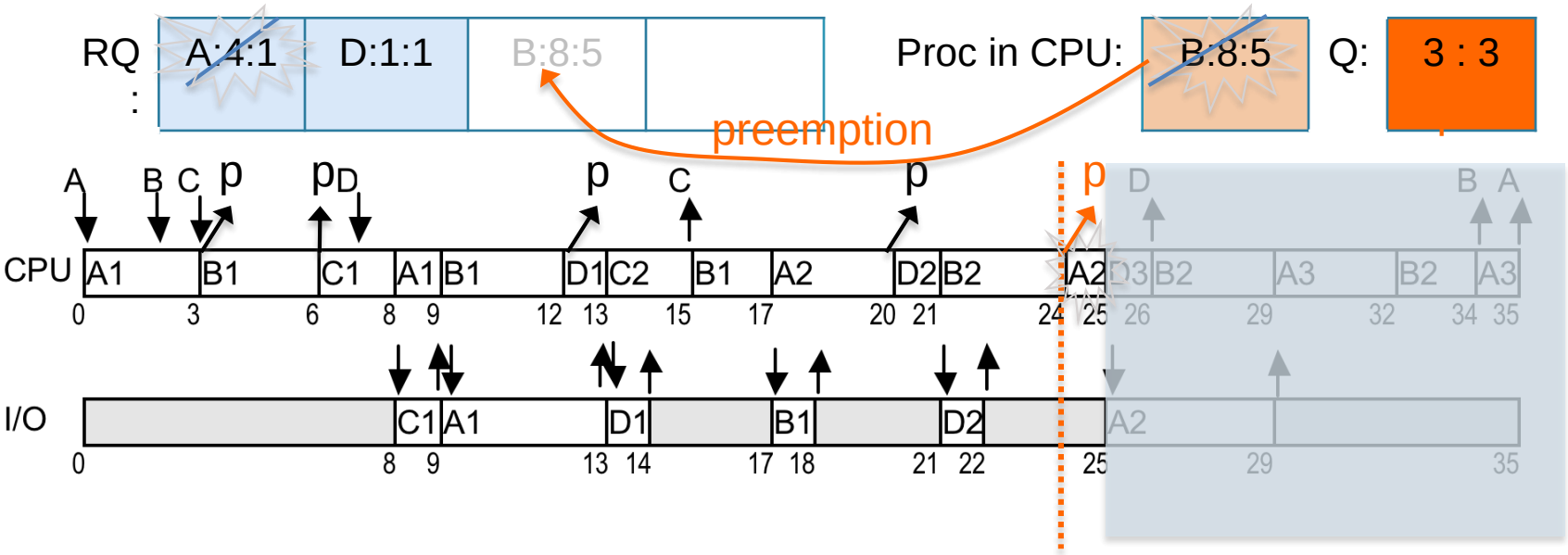
Proc in CPU: B:8:7

Q: 3 : 1 expires



Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 1	4	4 : 4
B	2	8 : 0	1	8 : 5	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 1

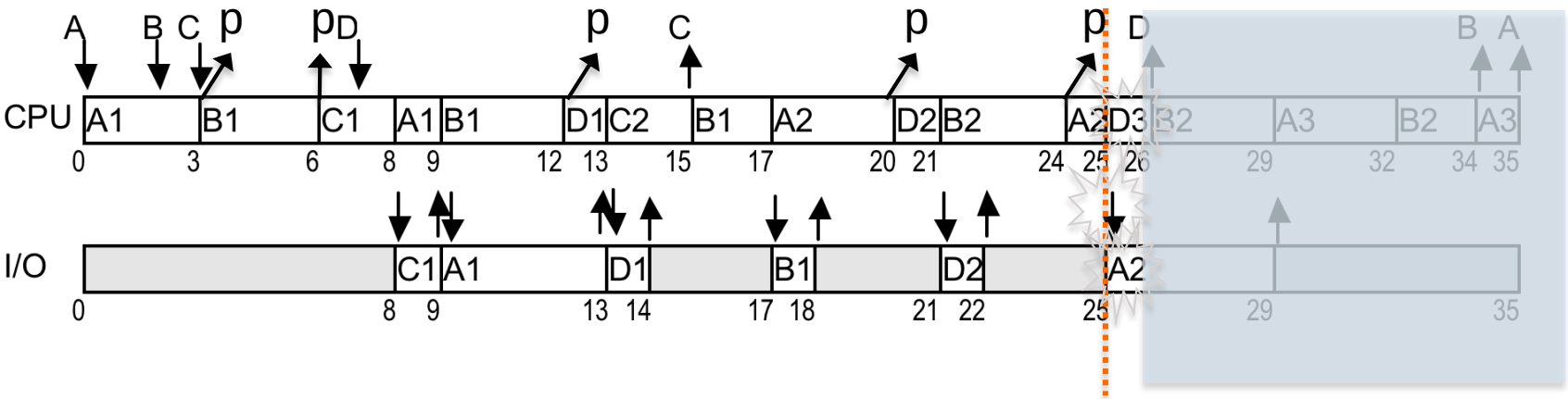


Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 4
B	2	8 : 0	1	8 : 5	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 1

RQ : ~~D:1:1~~ | B:8:5 | | |

Proc in CPU: | Q: 3 : 0



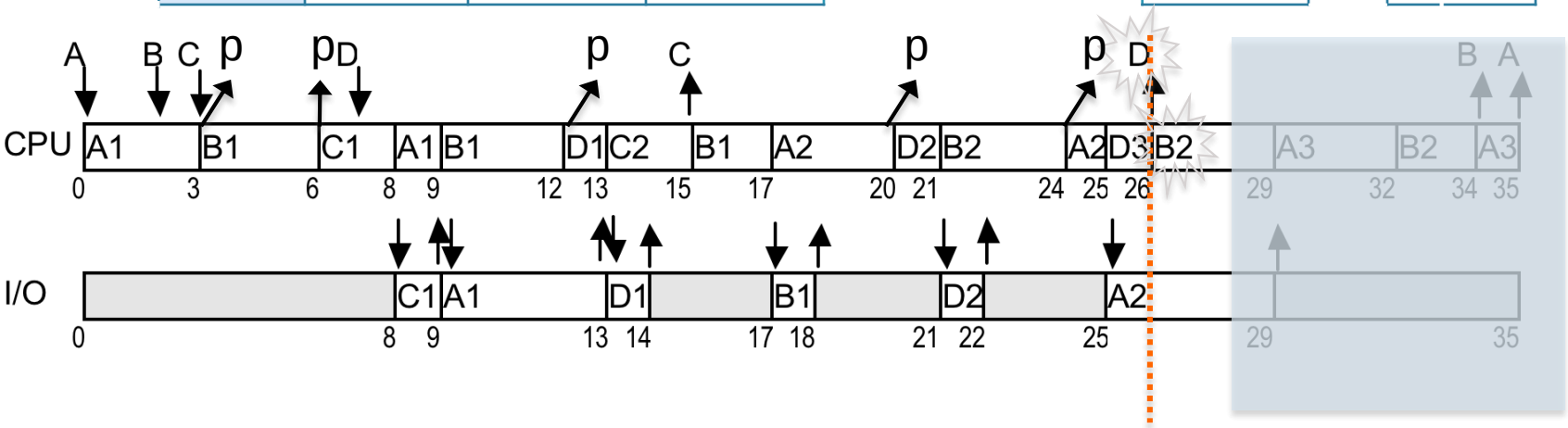
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 4
B	2	8 : 0	1	8 : 5	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

RQ : B:~~8:5~~

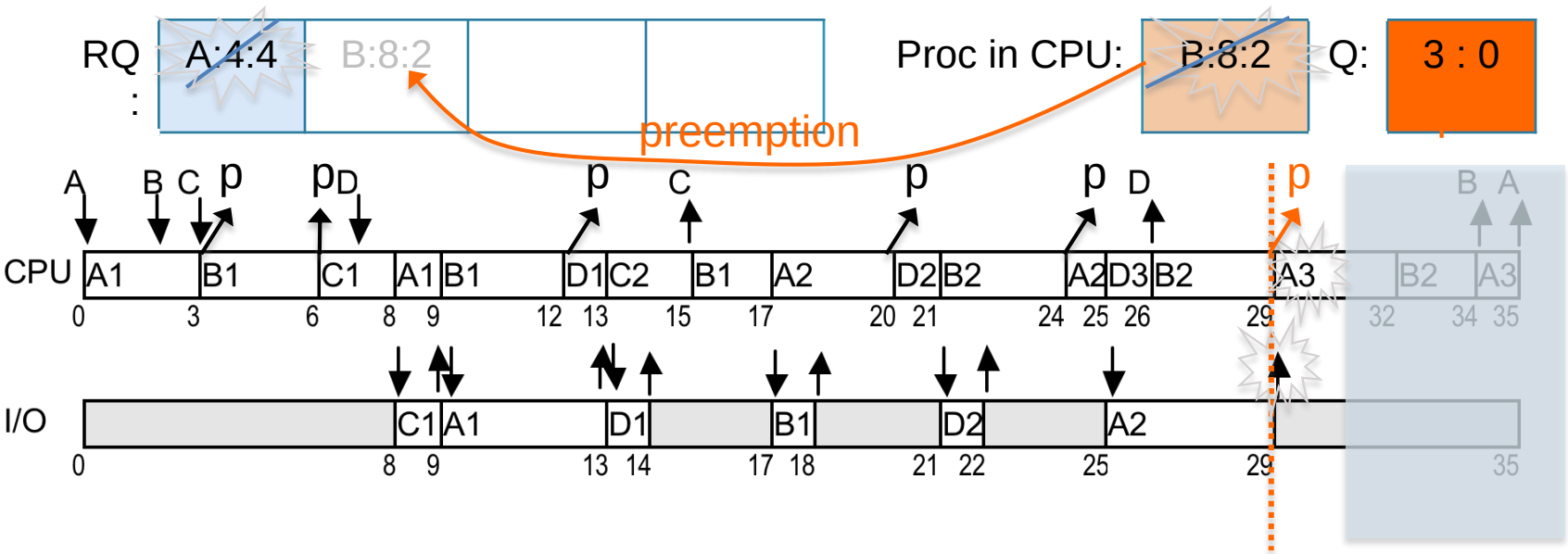
Proc in CPU:

Q: 3 : 0



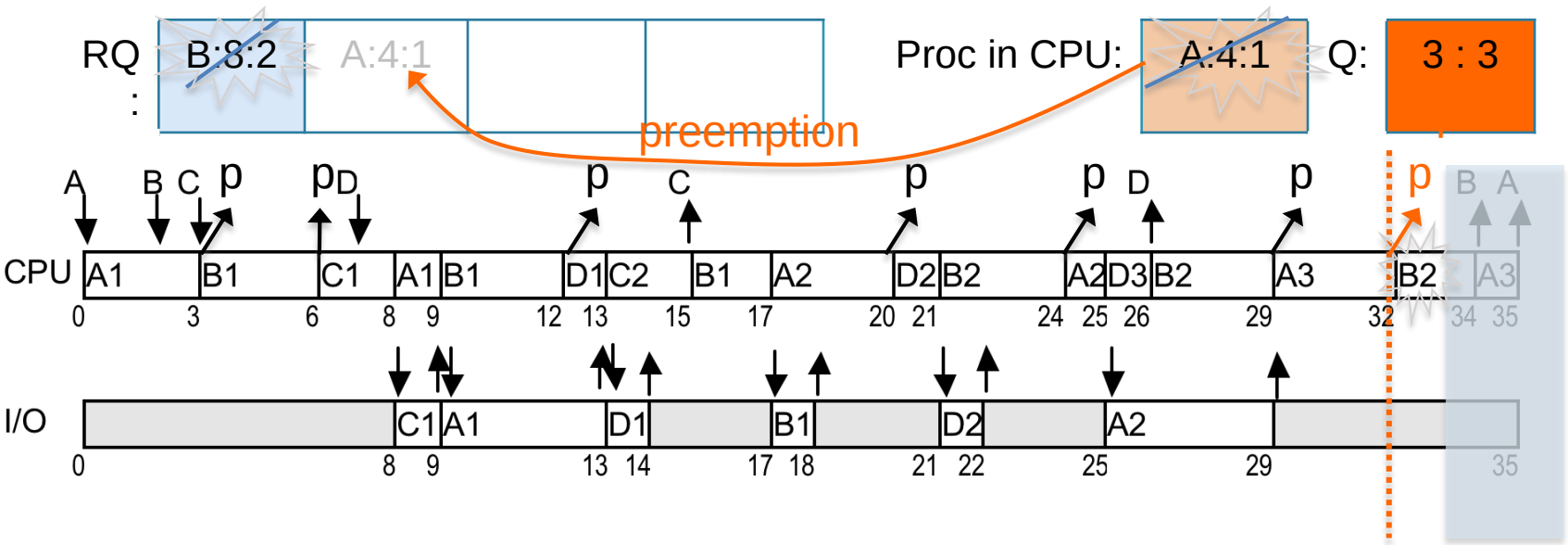
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 4
B	2	8 : 0	1	8 : 2	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0



Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 1
B	2	8 : 0	1	8 : 2	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0



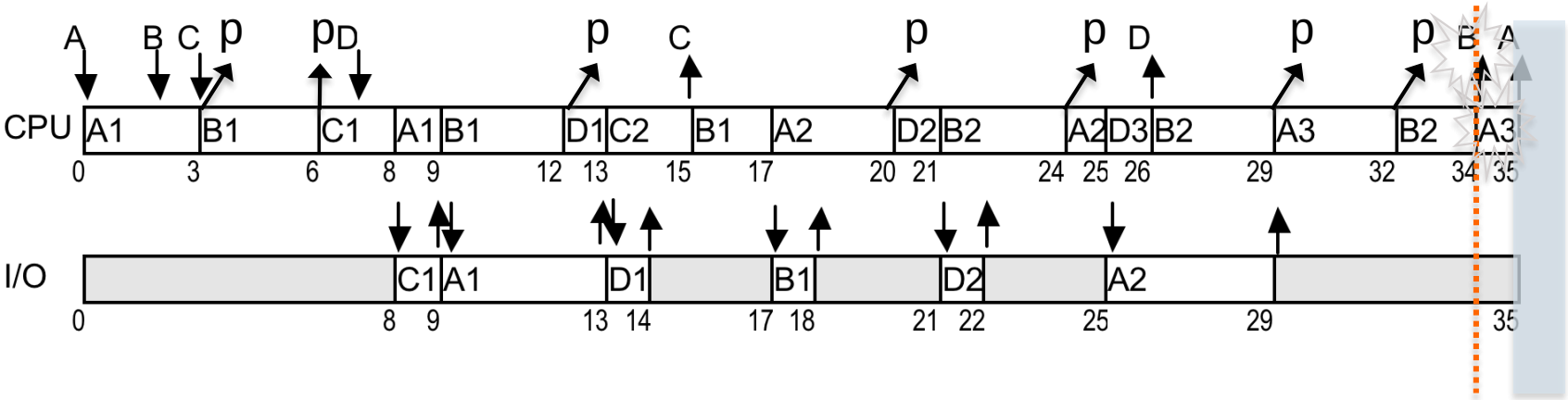
Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 1
B	2	8 : 0	1	8 : 0	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

RQ : A ~~4:1~~

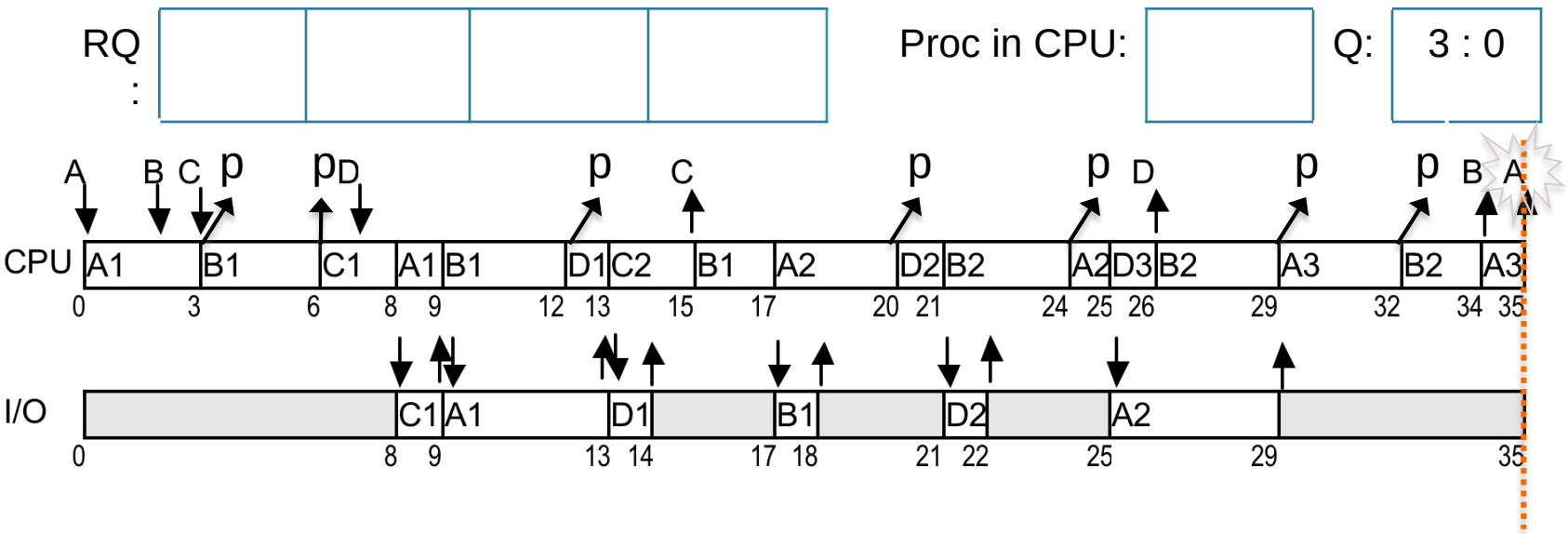
Proc in CPU:

Q: 3 : 0

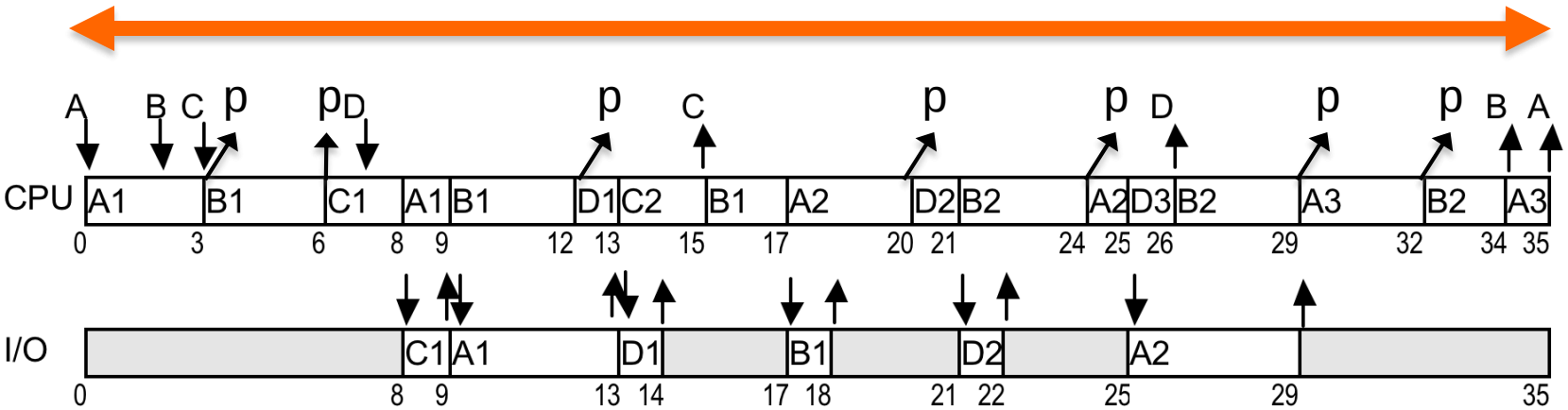


Round-Robin Scheduling (RRS)

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4 : 0	4	4 : 0	4	4 : 0
B	2	8 : 0	1	8 : 0	-	-
C	3	2 : 0	1	2 : 0	-	-
D	7	1 : 0	1	1 : 0	1	1 : 0

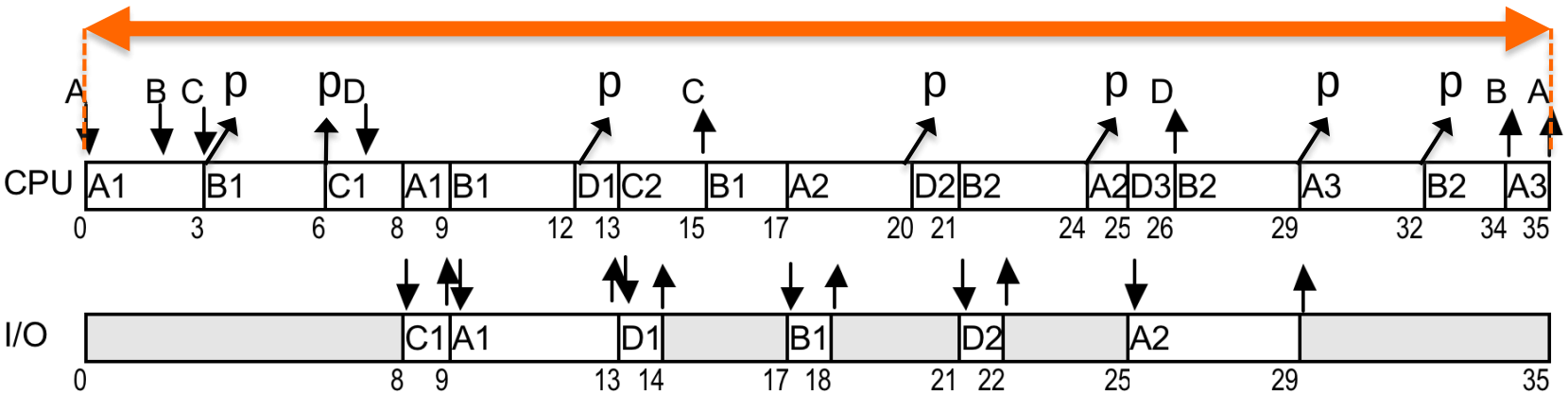


Round-Robin Scheduling (RRS)



- Processor utilization = (35 / 35) * 100 = 100 %
- Throughput = 4 / 35

Round-Robin Scheduling (RRS)



- Turn around time:

  $tat_A = 35 - 0 = 35$

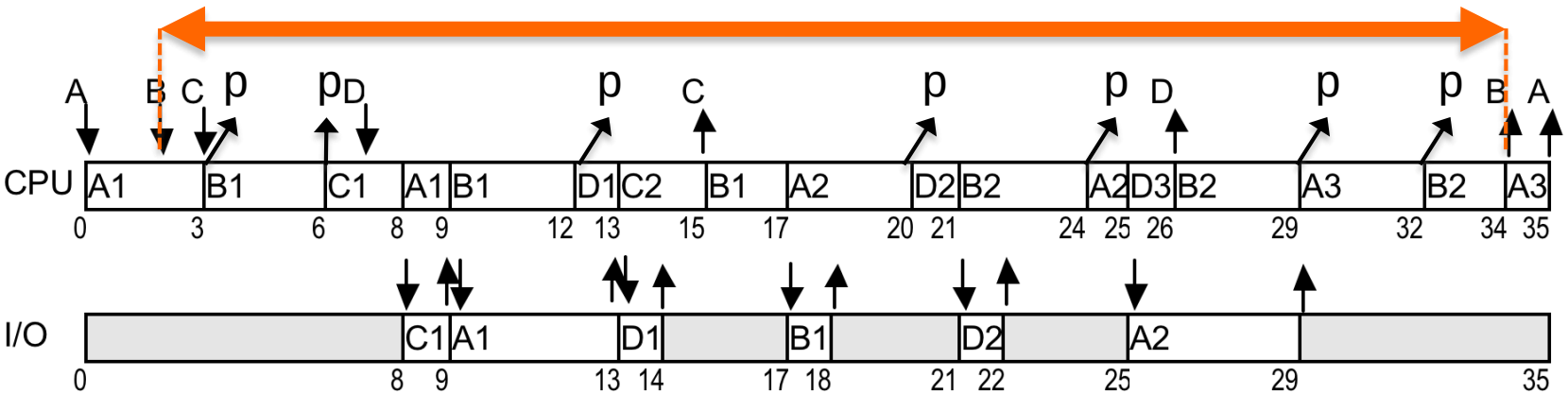
  $tat_B = 34 - 2 = 32$

  $tat_C = 15 - 3 = 12$

  $tat_D = 26 - 7 = 19$

  $tat_{AVG} = (35 + 32 + 12 + 19) / 4 = 24.$

Round-Robin Scheduling (RRS)



- Turn around time:

  $tat_A = 35 - 0 = 35$

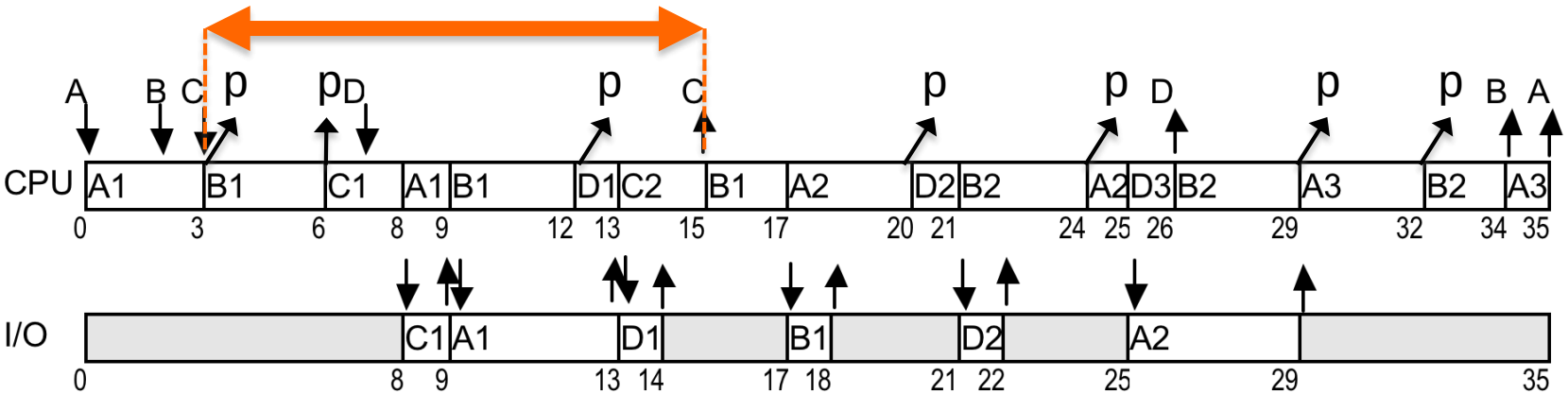
  $tat_B = 34 - 2 = 32$

  $tat_C = 15 - 3 = 12$

  $tat_D = 26 - 7 = 19$

  $tat_{AVG} = (35 + 32 + 12 + 19) / 4 = 24.$

Round-Robin Scheduling (RRS)



- Turn around time:

  $tat_A = 35 - 0 = 35$

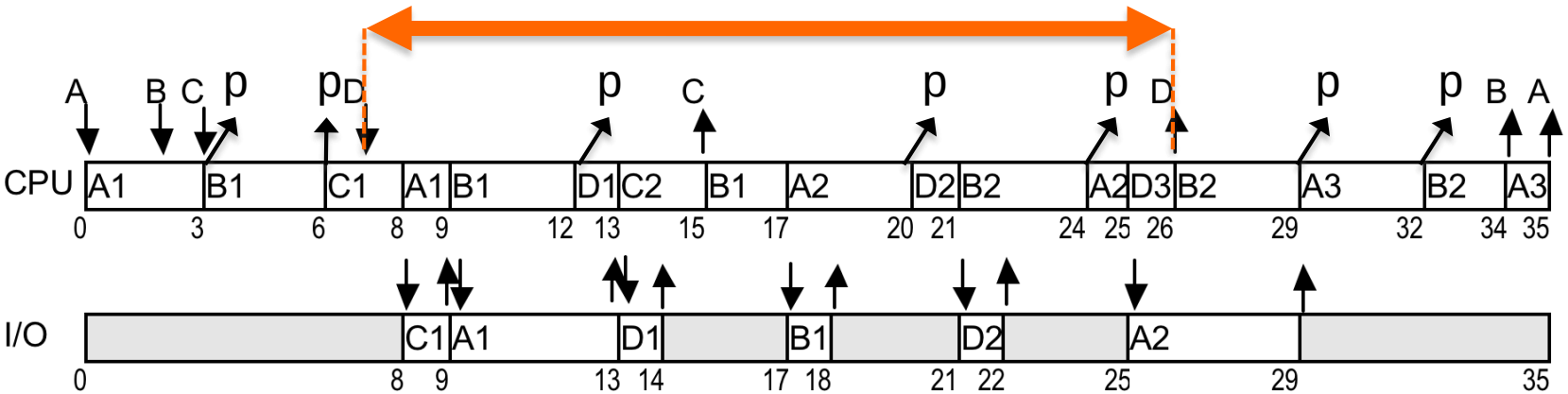
  $tat_B = 34 - 2 = 32$

  $tat_C = 15 - 3 = 12$

  $tat_D = 26 - 7 = 19$

  $tat_{AVG} = (35 + 32 + 12 + 19) / 4 = 24.$

2.3.4 Round-Robin Scheduling (RRS)



- Turn around time:

  $tat_A = 35 - 0 = 35$

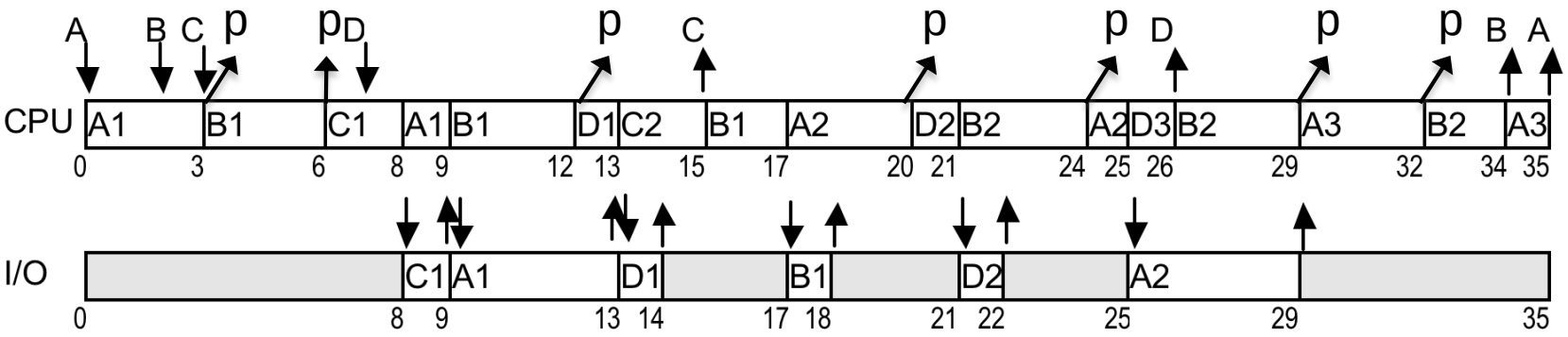
  $tat_B = 34 - 2 = 32$

  $tat_C = 15 - 3 = 12$

  $tat_D = 26 - 7 = 19$

  $tat_{AVG} = (35 + 32 + 12 + 19) / 4 = 24.$

Round-Robin Scheduling (RRS)



- Turn around time:

  $tat_A = 35 - 0 = 35$

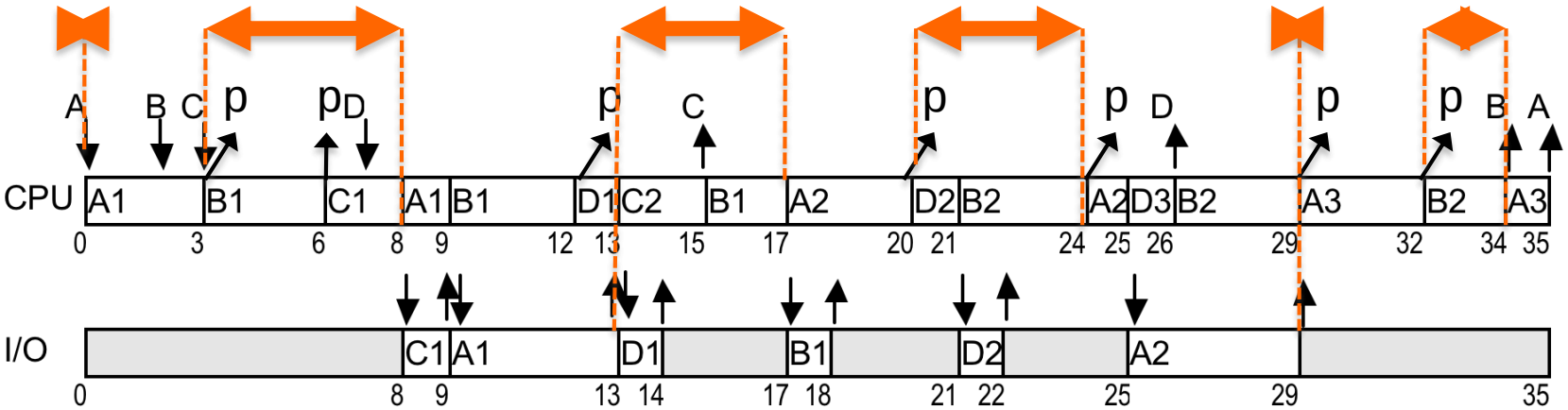
  $tat_B = 34 - 2 = 32$

  $tat_C = 15 - 3 = 12$

  $tat_D = 26 - 7 = 19$

  $tat_{AVG} = (35 + 32 + 12 + 19) / 4 = 24$

Round-Robin Scheduling (RRS)



- Waiting time:

  $wt_A = (0 – 0)+(8 – 3)+(17 – 13)+(24 – 20)+(29 – 29)+(34 – 32)=15$

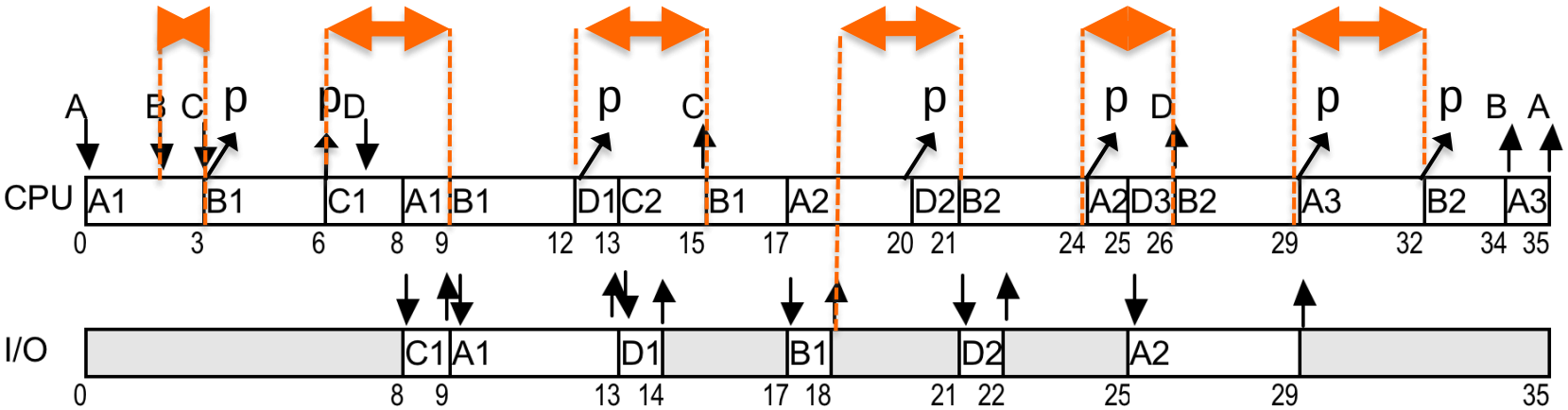
  $wt_B = (3 – 2)+(9 – 6)+ (15 -12)+(21 – 18)+(26 – 24)+(32 – 29) =15$

  $wt_C = (6 – 3) + (13 – 9) = 7$

  $wt_D = (12 – 7) + (20 – 14) + (25 – 22) = 14$

  $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

Round-Robin Scheduling (RRS)



- Waiting time:

  $wt_A = (0 – 0)+(8 – 3)+(17 – 13)+(24 – 20)+(29 – 29)+(34 – 32)=15$

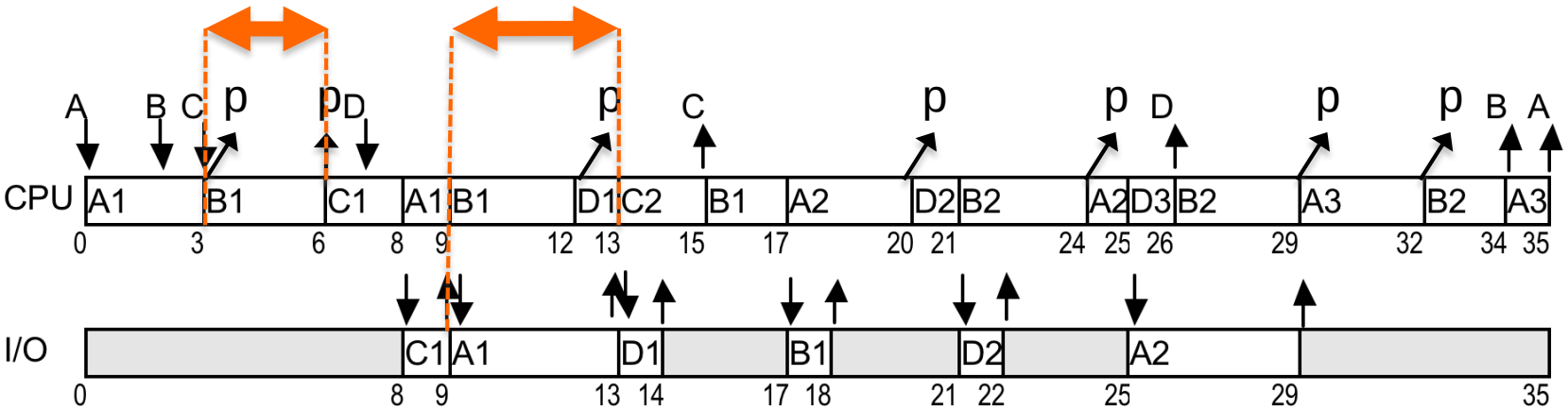
  $wt_B = (3 – 2)+(9 – 6)+ (15 -12)+(21 – 18)+(26 – 24)+(32 – 29) =15$

  $wt_C = (6 – 3) + (13 – 9) = 7$

  $wt_D = (12 – 7) + (20 – 14) + (25 – 22) = 14$

  $wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

Round-Robin Scheduling (RRS)



- Waiting time:

$wt_A = (0 – 0)+(8 – 3)+(17 – 13)+(24 – 20)+(29 – 29)+(34 – 32)=15$

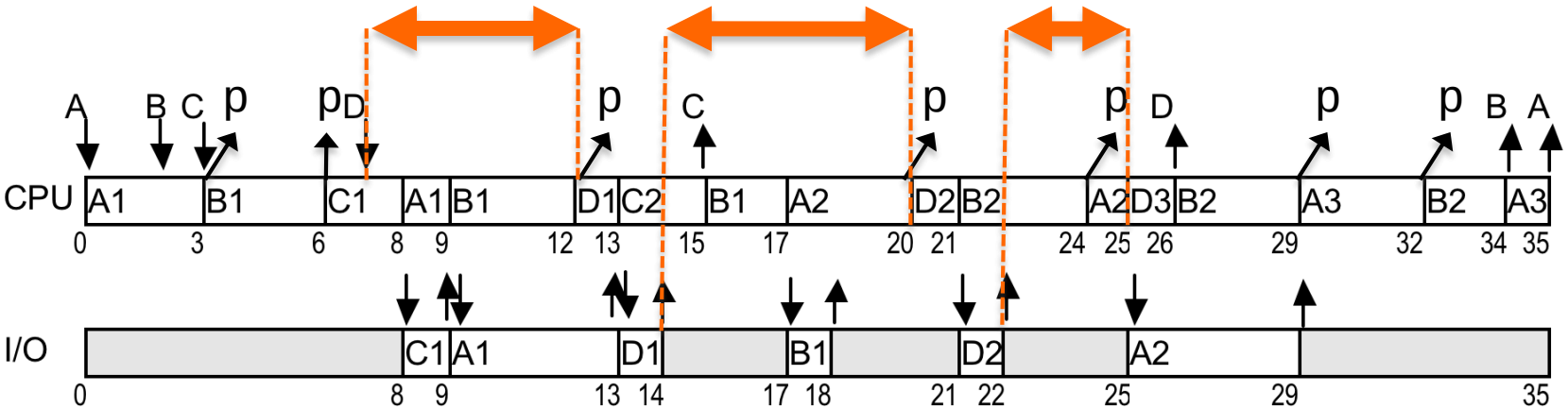
$wt_B = (3 – 2)+(9 – 6)+ (15 -12)+(21 – 18)+(26 – 24)+(32 – 29) =15$

$wt_C = (6 – 3) + (13 – 9) = 7$

$wt_D = (12 – 7) + (20 – 14) + (25 – 22) = 14$

$wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

Round-Robin Scheduling (RRS)



- Waiting time:

$wt_A = (0 – 0)+(8 – 3)+(17 – 13)+(24 – 20)+(29 – 29)+(34 – 32)=15$

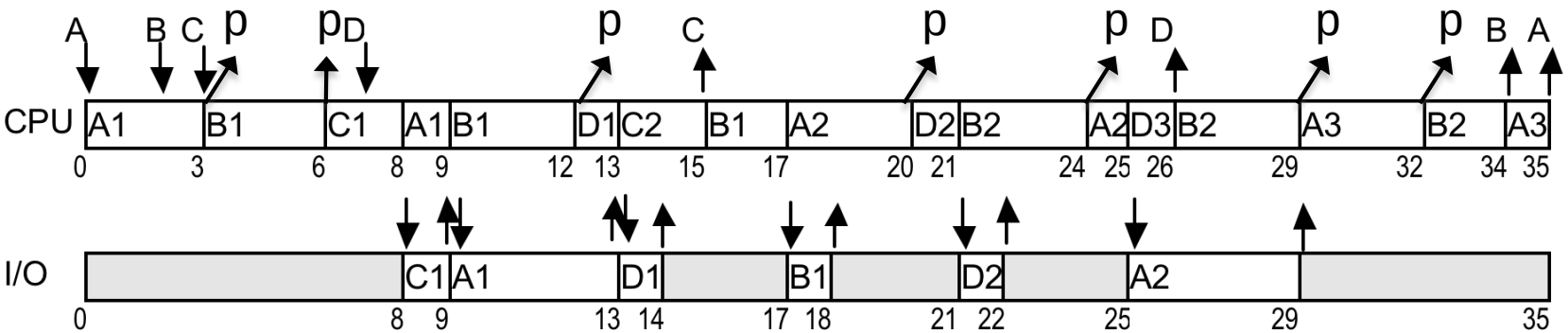
$wt_B = (3 – 2)+(9 – 6)+ (15 -12)+(21 – 18)+(26 – 24)+(32 – 29) =15$

$wt_C = (6 – 3) + (13 – 9) = 7$

$wt_D = (12 – 7) + (20 – 14) + (25 – 22) = 14$

$wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

Round-Robin Scheduling (RRS)



- Waiting time:

$wt_A = (0 – 0)+(8 – 3)+(17 – 13)+(24 – 20)+(29 – 29)+(34 – 32)=15$

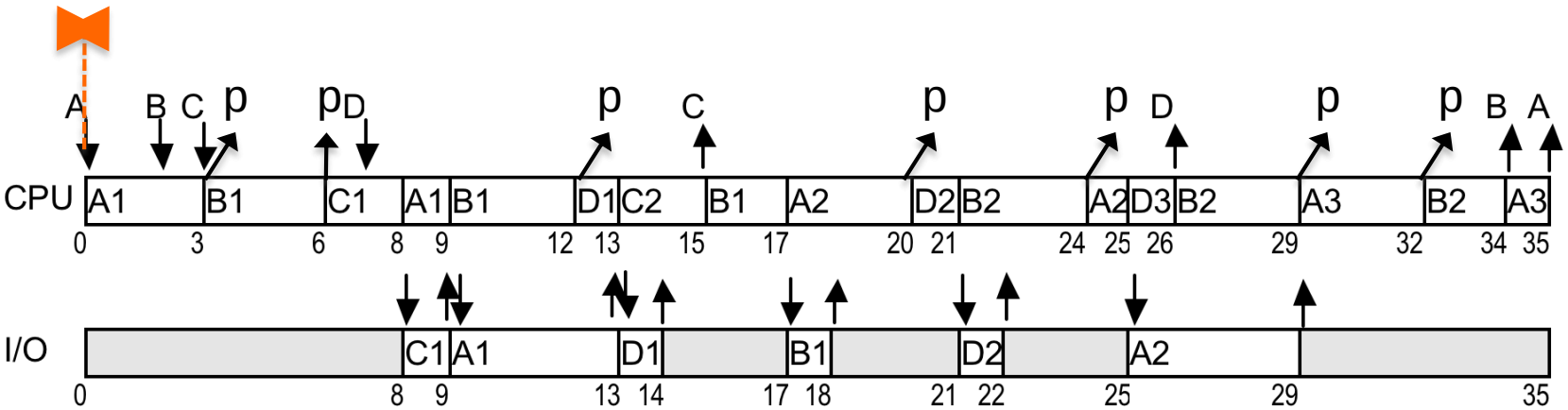
$wt_B = (3 – 2)+(9 – 6)+ (15 -12)+(21 – 18)+(26 – 24)+(32 – 29) =15$

$wt_C = (6 – 3) + (13 – 9) = 7$

$wt_D = (12 – 7) + (20 – 14) + (25 – 22) = 14$

$wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

Round-Robin Scheduling (RRS)



- Response time:

  $rt_A = 0 - 0 = 0$

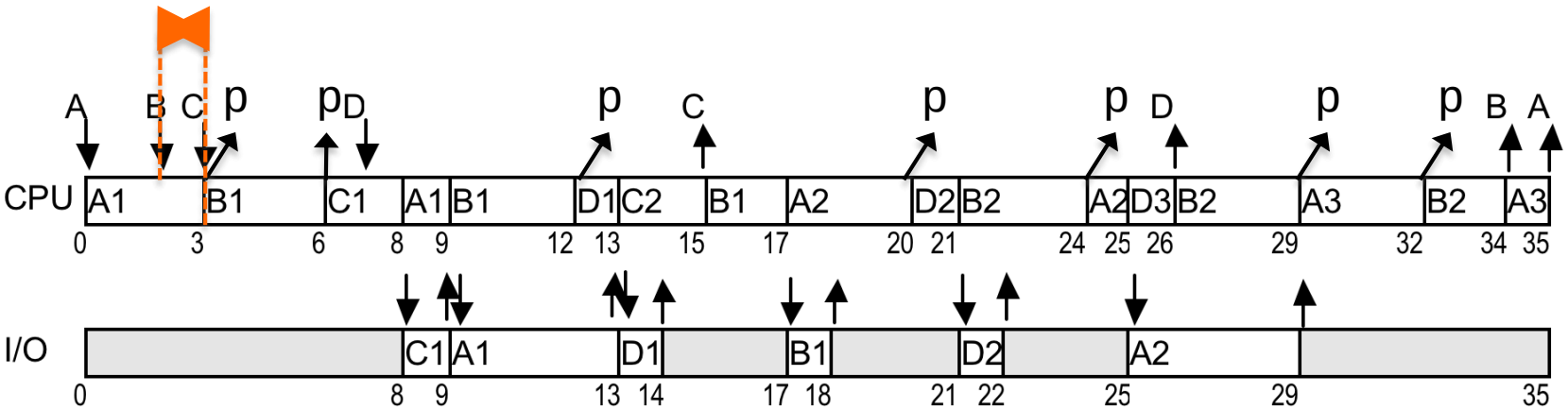
  $rt_B = 36 - 2 = 1$

  $rt_C = 6 - 3 = 3$

  $rt_D = 12 - 7 = 5$

  $rt_{AVG} = (0 + 1 + 3 + 5) / 4 = 2.25$

Round-Robin Scheduling (RRS)



- Response time:

$rt_A = 0 - 0 = 0$

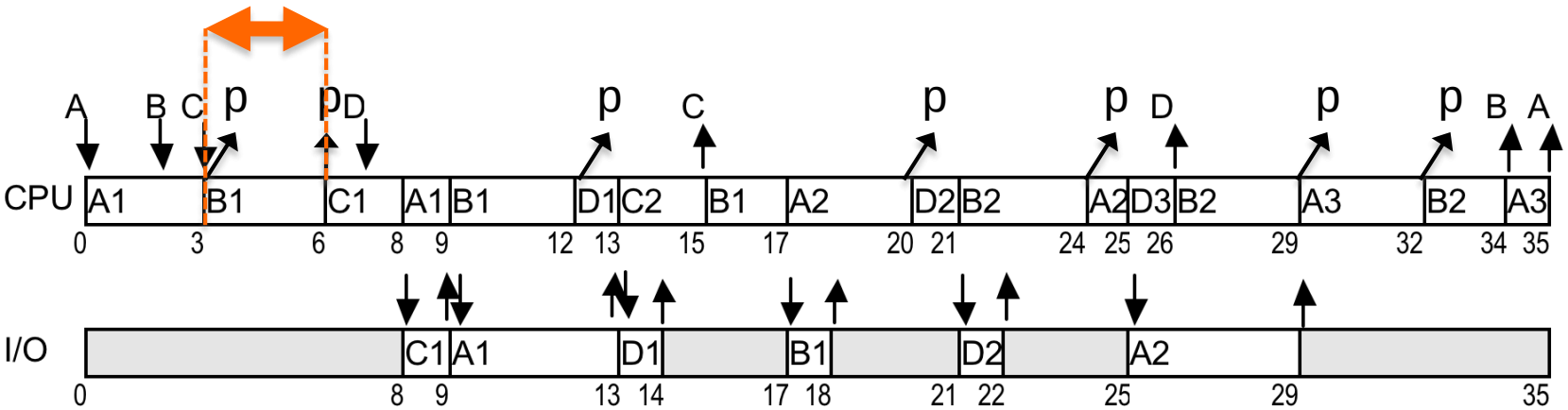
$rt_B = 3 - 2 = 1$

$rt_C = 6 - 3 = 3$

$rt_D = 12 - 7 = 5$

$rt_{AVG} = (0 + 1 + 3 + 5) / 4 = 2.25$

Round-Robin Scheduling (RRS)



- Response time:

  $rt_A = 0 - 0 = 0$

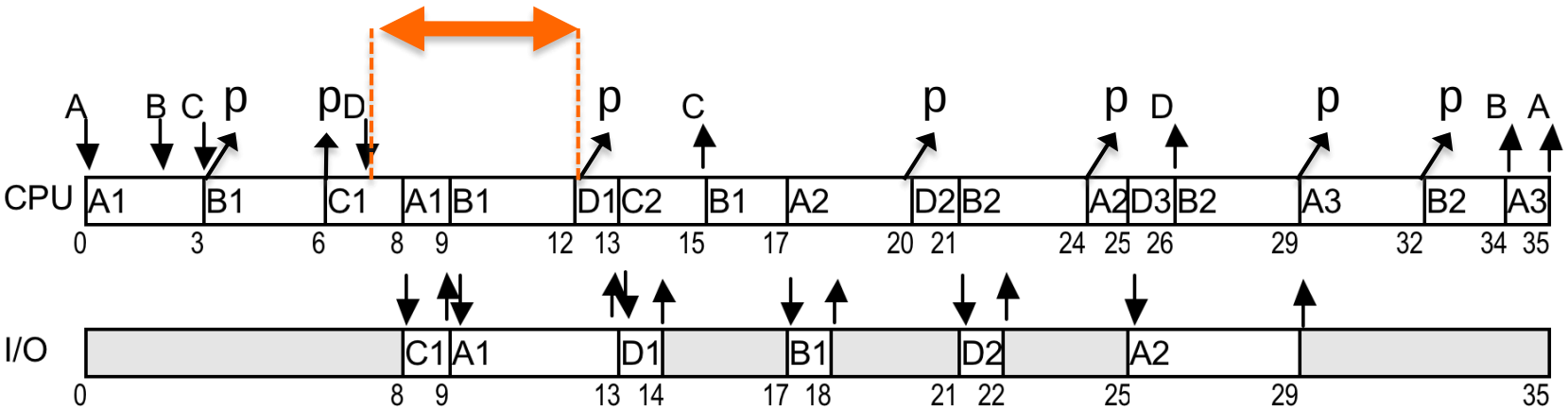
  $rt_B = 36 - 2 = 1$

  $rt_C = 6 - 3 = 3$

  $rt_D = 12 - 7 = 5$

  $rt_{AVG} = (0 + 1 + 3 + 5) / 4 = 2.25$

Round-Robin Scheduling (RRS)



- Response time:

$rt_A = 0 - 0 = 0$

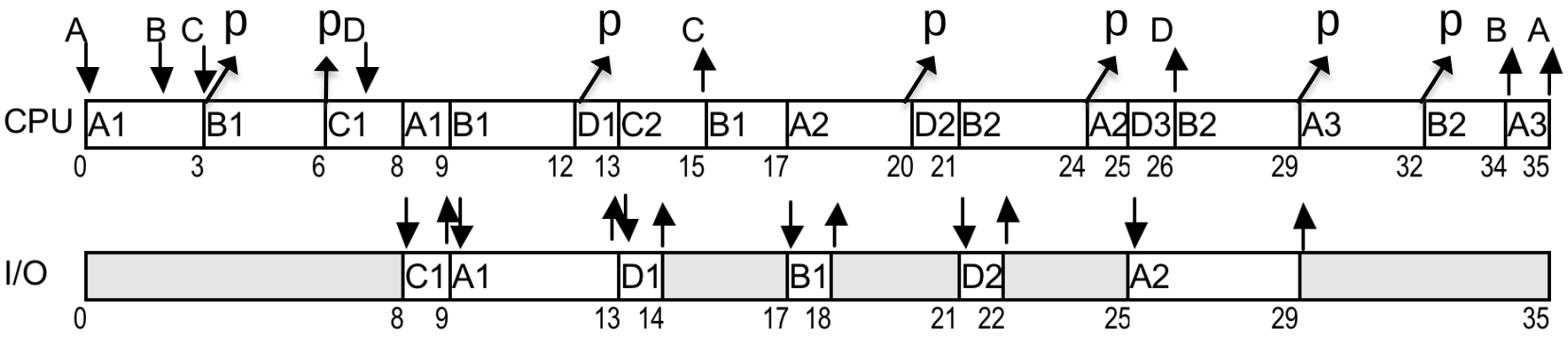
$rt_B = 36 - 2 = 1$

$rt_C = 6 - 3 = 3$

$rt_D = 12 - 7 = 5$

$rt_{AVG} = (0 + 1 + 3 + 5) / 4 = 2.25$

Round-Robin Scheduling (RRS)



- Response time:

  $rt_A = 0 - 0 = 0$

  $rt_B = 3 - 2 = 1$

  $rt_C = 6 - 3 = 3$

  $rt_D = 12 - 7 = 5$

  $rt_{AVG} = (0 + 1 + 3 + 5) / 4 = 2.25$

Round-Robin Scheduling (RRS)

- In Round Robin Scheduling it is guaranteed that

  $rt_{max} \leq (n-1)*Q$

  where

  n : multiprogramming degree

  Q: time quantum

comparison

	FCFS	SPF	SRTF	RR
tat_{avg}	28.25	23.50	18.75	24.50
wt_{avg}	16.50	10.50	6.25	12.25
rt_{avg}	4.50	3.00	1.25	2.25
	Easy to implement	Not possible to know next CPU burst exactly, it can only be guessed	Not possible to know next CPU burst exactly, it can only be guessed	Implementable, rt_{max} is important for interactive systems

Priority Scheduling

- In this type of algorithms a priority is associated with each process and the processor is given to the process with the highest priority. Equal priority processes are scheduled with FCFS method.

- To illustrate, SPF is a special case of priority scheduling algorithm where

Priority(i) = 1 / next processor burst time of process i

or

Priority(i) = K - next processor burst time of process i

Priority Scheduling

- Priorities can be fixed externally or they may be calculated by the OS from time to time.
- Externally, if all users have to code time limits and maximum memory for their programs, priorities are known before execution.
- Internally, a next processor burst time prediction such as that of SPF can be used to determine priorities dynamically.

Priority Scheduling

- A priority scheduling algorithm can leave some low-priority processes in the ready queue indefinitely.
- If the system is heavily loaded, it is highly probable that there is a higher-priority process to grab the processor.
- This is called *starvation.*
- One solution for the starvation problem might be to gradually increase the priority of processes that stay in the system for a long time.

Priority Scheduling

Example: Following may be used as a priority defining function:

Priority (i) = $10 + t_{now} - t_s(i) - t_{rq}(i) - t_{cpu}(i)$

where

$t_s(i)$: the time process *i* is submitted to the system

$t_{rq}(i)$: the time process *i* entered to the ready queue last time

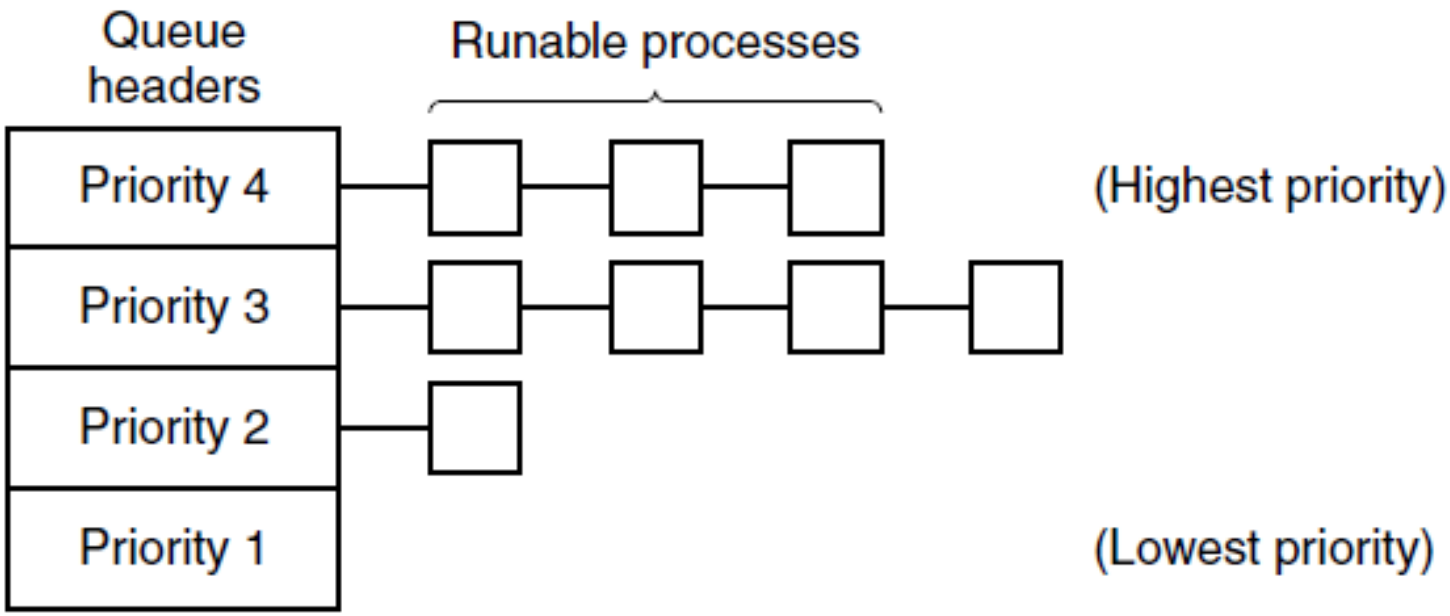
$t_{cpu}(i)$: next processor burst length of process *i*

t_{now}: current time

- $P(i)=t_{now} - t_{rq}(i) \approx$FCFS
- $P(i)=K - t_{cpu}(i) \approx$SPF
- $P(i)=\alpha(t_{now} - t_{rq}(i))+(1-\alpha)(K - t_{cpu}(i))$,

$\approx$combination of FCFS & SPF

Priority Scheduling



A scheduling algorithm with four priority classes.

Guaranteed Scheduling

- Makes real promises to users about performance.
- One promise is: If n users are logged in while you are working, you will receive about 1/n of the CPU power.
- Similarly, on a single-user system with n processes running, all things being equal, each one should get 1/n of the CPU cycles.
  - System keeps track of how much CPU each process has had since its creation.
  - It computes compute the ratio of actual CPU time consumed to CPU time entitled.
  - The algorithm is then to run the process with the lowest ratio until its ratio has moved above that of its closest competitor.
  - Then that one is chosen to run next.

Scheduling in Real-Time Systems

- Time plays an essential role
- Categories
  - Hard real time
  - Soft real time
  - Periodic or aperiodic
- A schedulable case satisfies $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$

  where
  - m periodic events exist and event i occurs with period Pi and requires Ci sec of CPU time to handle each event

<u>Example</u>: consider three periodic events,

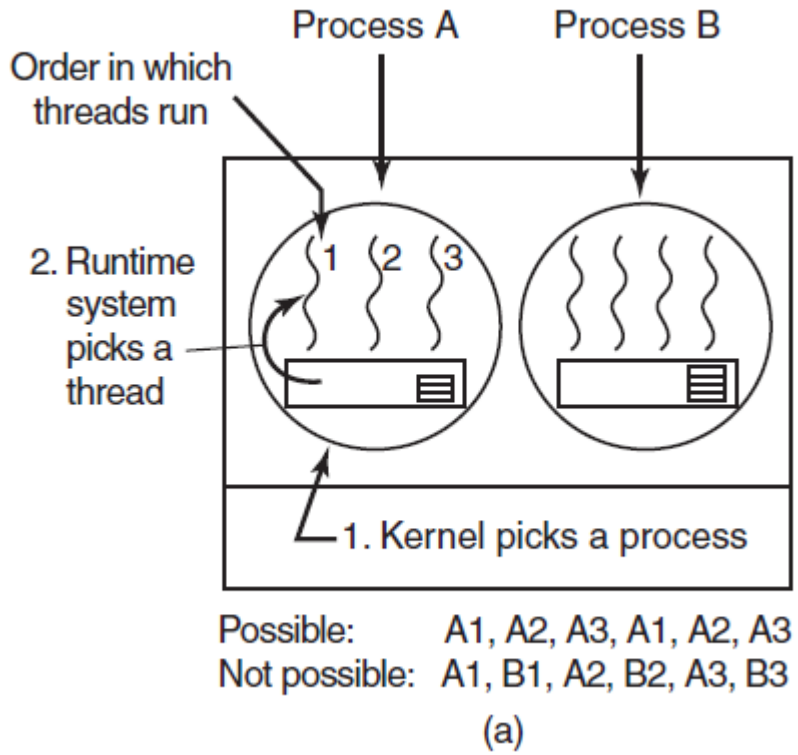
periods: 100, 200, and 500 msec, and

cpu time: 50, 30, and 100 msec, respectively,

system is schedulable 0. 5 + 0. 15 + 0. 2 < 1.

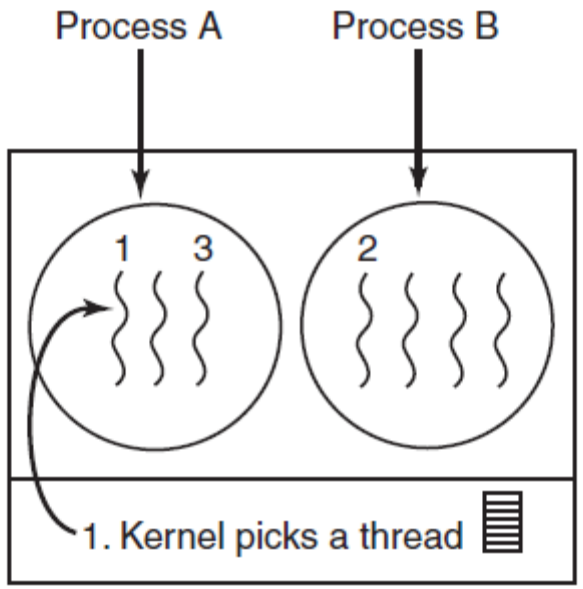
If a fourth event with a period of 1 sec is added, it will remain schedulable as long as it does not need more than 150 msec of CPU time per event.

Thread Scheduling (1)



Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst.

Thread Scheduling (2)



Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

(b)

Possible scheduling of kernel-level threads with the same characteristics as (a).

End

Chapter 2