CS-202 Exercises on I/O and Scheduling (L08 - L09)

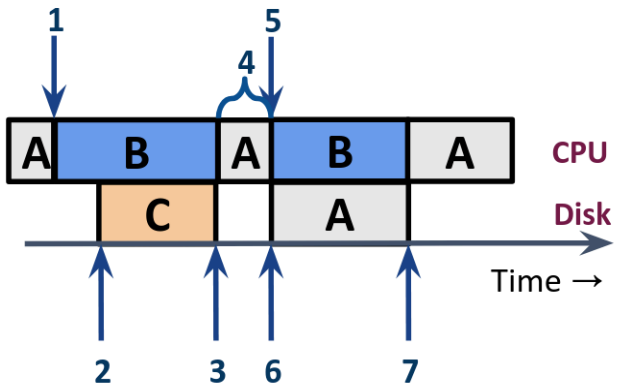
24.03.2025

This exercise set covers concepts related to I/O and scheduling. We advise that you work through it sequentially, referring back to lecture slides or videos as necessary. If anything is unclear, or if you could benefit from discussing a particular concept in depth, please seek an assistant's help.

Exercise 1: CPU-device communication through interrupts

The CPU interacts with I/O devices through interrupts: When the CPU runs a thread that needs I/O, the CPU requests an I/O operation and switches to another thread, while waiting for the I/O operation to complete. When the device completes the operation, it generates an interrupt; the OS handles the interrupt by processing the data and resuming the original thread (or setting its status to Ready).

The diagram below illustrates this interaction in the particular scenario where the device is a disk. Letters A, B, and C represent threads. Numbers 1, 4, and 5 represent events initiated by the CPU, whereas numbers 2,3,6, and 7 represent events initiated by the disk. The diagram shows which thread the CPU and the disk are working for at each point in time. E.g., before event 1 the CPU is working for thread A, but after event 1 it is working for thread B.



Connect each event on the left to a description on the right:

1.

2.

3.

4.

5.

6.

7.

a. The device driver (operating on behalf of thread A) copies the data to a memory-mapped or IO-mapped data register.

b. The disk sends an interrupt to the CPU, signaling that it has completed the requested I/O operation.

c. Thread A makes a `write()` system call. The OS puts the thread in the Blocked state

d. The disk begins the requested I/O operation

e. The device driver writes a command to a control register to initiate the I/O operation.

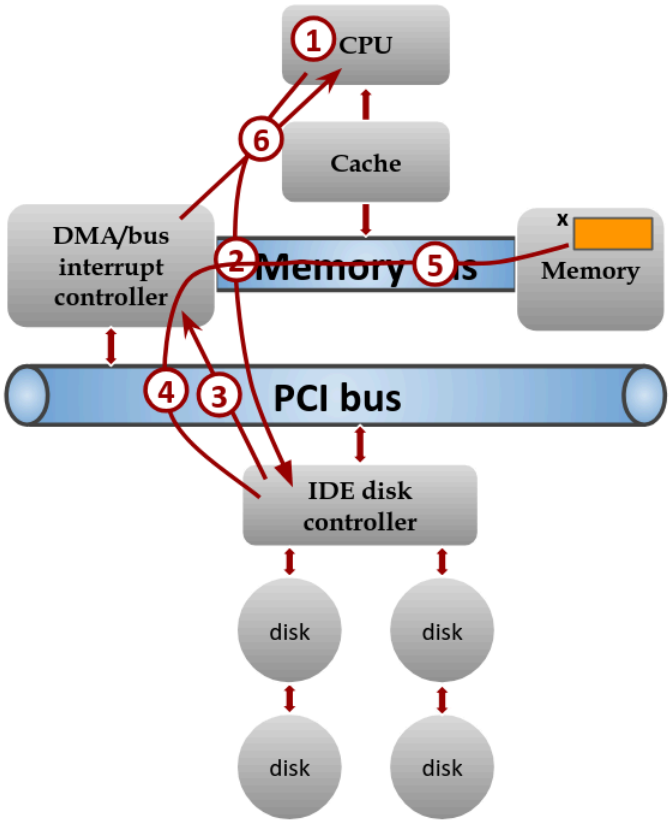
f. The disk completes the previous I/O operation and sends an interrupt to signal the CPU that it is now available.

Solution:

1. c – Thread A makes a `write()` system call. The OS puts the thread in the Blocked state
2. d – The disk begins the requested I/O operation
3. f – The disk completes the previous I/O operation and sends an interrupt to signal the CPU that it is now available.
4. a – The device driver (operating on behalf of Thread A) copies the data to a memory-mapped or IO-mapped data register.
5. e – The device driver writes a command to a control register to initiate the I/O operation.
6. d – The disk begins the requested I/O operation
7. b – The disk sends an interrupt to the CPU, signaling that it has completed the requested I/O operation.

Exercise 2: Direct Memory Access (DMA)

The diagram below illustrates data transfer between disk and memory using Direct Memory Access (DMA).



Connect each step on the left to a description on the right:

1.
2.
3.
4.
5.
6.

a. CPU tells device driver to transfer disk data to buffer at address X

b. When C = 0, DMA interrupts CPU to signal transfer completion

c. DMA controller transfers bytes to buffer X, increasing memory address and decreasing until C = 0

d. Disk controller sends each byte to DMA controller

e. Device driver tells disk controller to transfer C bytes from disk to buffer at address X

f. Disk controller initiates DMA transfer

Solutions

1. a. CPU tells device driver to transfer disk data to buffer at address X
2. e. Device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. f. Disk controller initiates DMA transfer
4. d. Disk controller sends each byte to DMA controller
5. c. DMA controller transfers bytes to buffer X, increasing memory address and decreasing until C = 0
6. b. When C = 0, DMA interrupts CPU to signal transfer completion

Exercise 3: I/O operations and system calls

The following is a C program that reads a text file (`text.txt`), counts the number of lines and vowels, and then writes the results to an output file `out.txt`.

```c
#include <stdio.h>
#include <ctype.h>

#define BUFFER_SIZE 4096

// Function to count vowels in a buffer
int count_vowels(const char *buffer, size_t size) {
    int count = 0;
    for (size_t i = 0; i < size; i++) {
        if (strchr("AEIOUaeiou", buffer[i])) {
            count++;
        }
    }
    return count;
}

int main() {
    char buffer[BUFFER_SIZE];
    int total_lines = 0, total_vowels = 0;

    FILE *src = fopen("text.txt", "r");

    size_t bytes_read;
    while ((bytes_read = fread(buffer, 1, BUFFER_SIZE, src)) > 0) {
        total_vowels += count_vowels(buffer, bytes_read);
        for (size_t i = 0; i < bytes_read; i++) {
            if (buffer[i] == '\n') total_lines++; // Count lines
        }
    }
    fclose(src);

    FILE *dest = fopen("out.txt", "w");
    if (!dest) { perror("Error opening file"); return 1; }
    fprintf(dest, "Lines: %d\nVowels: %d\n", total_lines, total_vowels);
    fclose(dest);

    return 0;
}
```

Questions:

1. Identify the calls that result in I/O operations and the corresponding syscalls (if any). Write each call to a cell in the first column of the table below.

2. Does your answer to the above question change if `text.txt` is already in the file system buffer cache?
3. For each call mentioned in your answer to Question 1, indicate which system layers it "touches" by adding an X to the corresponding cell of the same row..

Operation	Application	Library	File system	Block device interface	Physical device
`fopen`					

Solutions:

Operation	Application	Library	File system	Block device interface	Physical device	Explanation
`fopen("text.txt", "r")`	X	X	X	X	X (unless cached)	Open a file for reading. Internally uses `open()` syscall. Touch everything down to disk unless cached.
`fread(buffer, ...)`	X	X	X	X	X (unless cached)	Reads file contents into memory using `read()` syscall.

Operation	Application	Library	File system	Block device interface	Physical device	Explanation
						Touch everything down to disk unless cached.
`fclose(src)`	X	X	X			Flushes and closes file descriptor. Only affects file system metadata.
`fopen("out.txt", "w")`	X	X	X	X	X	Opens/creates file for writing. Touches all layers including disk for inode + file allocation.
`fprintf(dest, ...)`	X	X	X	X	X	Writes to the file with `write()` system call.
`fclose(dest)`	X	X	X	X	X	Flushes the output buffer to file using `write()`, closes the file descriptor.
`perror(...)` (optional, unless out.txt doesn't exist)	X	X	X	X	X	Optional – only runs if `fopen(out.txt)` fails. Writes an error message to stderr.

Exercise 4: Context switching

Consider the following program:

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int pid = fork();

    if (pid == 0) {
        // Child process
        sleep(1);
    } else if (pid > 0) {
        // Parent process
        printf("Parent process\n");
        waitpid(pid, NULL, 0);
    }

    return 0;
}
```

Questions:

1. What is the state of the child process:
    a. Right before it calls `sleep(1)`
    b. Right after it calls `sleep(1)`
    c. After it finishes `sleep(1)`
2. Identify the calls that result in a context switch and list them in the first column of the table below. Next to each call, indicate what triggers the context switch (system call, I/O, or voluntary yield).

Call	Trigger Type

Solutions:

1. What is the state of the child process:
    a. Right before it calls `sleep(1)`: Running
    b. Right after it calls `sleep(1)`: Blocked
    c. After it finishes `sleep(1)`: Ready

2. Identify the calls that result in a context switch and list them in the first column of the table below. Next to each call, indicate what triggers the context switch (system call, I/O, or voluntary yield).

Call	Trigger Type
`fork()`	**System call**
`sleep(1)`	**Voluntary yield**
`waitpid(...)`	**System call**
`printf(...)`	**I/O***

*In reality, `printf()` **does not** necessarily cause a context switch unless it blocks (e.g., full output buffer or slow terminal), which isn't guaranteed.

Exercise 5: Basic questions on scheduling

1. Why might a thread not give up the CPU even when other threads are waiting to run? (Give 2 reasons)
2. How does the OS reduce the risk of this situation happening and ensure it can take back control when needed?
3. Why is it important for a scheduler to distinguish between threads that mostly perform CPU-intensive computation (“CPU-bound”threads) and threads that mostly perform I/O (“I/O-bound” threads)?

Solutions:

1. Why might a thread not give up the CPU even when other threads are waiting to run? (Give 2 reasons)
    1) It’s poorly written or buggy and never gives up.
    2) It’s doing long CPU-intensive work without giving up.
    3) It’s malicious and intentionally refuses to yield to starve other threads

2. How does the OS reduce the risk of this situation happening and ensure it can take back control when needed?
    1) Uses preemptive scheduling with timer interrupts.
    2) Takes back control automatically when a time slice ends.

3. Why is it important for a scheduler to distinguish between threads that mostly perform CPU-intensive computation (“CPU-bound”threads) and threads that mostly perform I/O (“I/O-bound” threads)?
    1) To improve responsiveness (I/O-bound threads get quick CPU bursts).
    2) To maximize CPU usage while I/O-bound threads are blocked.

Exercise 6: FIFO and SJF scheduling

1. Give one example of a First In First Out (FIFO) scheduler from daily life.
2. Consider the following thread arrival pattern:

Arrival Time	Length
0	10
1	2
5	20
11	5

Answer the following questions, first assuming a FIFO scheduler, then a Shortest Job First (SJF) scheduler:

a. When will the CPU finish executing all threads?
b. What is the average turnaround time?
c. What is the average response time?

Solution:

1. For example, waiting in line at a student restaurant

FIFO

Thread	Start	End	Turnaround Time = End - Arrival	Response Time = Start - Arrival
T1	0	10	10	0
T2	10	12	11	9

T3	12	32	27	7
T4	32	37	26	21

a. Finish time: 37

b. Average turnaround time: (10 + 11 + 27 + 26) / 4 = 18.5

c. Average response time: (0 + 9 + 7 + 21) / 4 = 9.25

SJF

Thread	Start	End	Turnaround Time = End - Arrival	Response Time = Start - Arrival
T1	0	10	10	0
T2	10	12	11	9
T4	12	17	6	1
T3	17	37	32	12

a. Finish time: 37

b. Average turnaround time: (10 + 11 + 6 + 32) / 4 = 14.75

c. Average response time: (0 + 9 + 1 + 12) / 4 = 5.5

Exercise 7: MLFQ Scheduling

Consider a Multi-Level Feedback Queue (MLFQ) scheduler with the following properties:

- Three priority levels, with the following time slices:
  - Level 1 (Highest Priority): 1 tick per time slice
  - Level 2 (Medium Priority): 2 ticks per time slice
  - Level 3 (Lowest Priority): 4 ticks per time slice

- Priority adjustment rules**:**
  - When a new thread arrives, it starts in Level 1.
  - If a thread uses up its time slice, it moves to one level below.
  - Every 10 ticks (so, at tick 10, 20, 30, etc) all threads are boosted to Level 1.
  - If a thread is blocked waiting for I/O, it does not consume CPU time and resumes execution after the I/O operation completes.

Consider the following arrival pattern: :

Thread ID	Arrival Time	Length
0	0	7
1	0	3
2	3	6
3	3	0.99 + I/O for 4 + 5

Questions:

1. For each time tick, which thread is executed and at which priority level?
   a. When does thread 3 resume execution after being blocked for I/O?
   b. How does priority boosting after 10 ticks affect the execution order?
   c. How does MLFQ balance short threads vs. long-running threads?

2. If a thread repeatedly performs I/O and avoids long CPU bursts, how does MLFQ treat it compared to CPU-bound threads?

Solutions

1. For each time tick, which thread is executed and at which priority level?

Tick	Thread Running	Priority Level	Notes
0	0	1	Thread 0 starts
1	1	1	Thread 1 starts
2	0	2	
3	0	2	
4	2	1	Thread 2 starts
5	3	1	Thread 3 starts (will enter I/O)
6	1	2	
7	1	2	T1 finishes
8	2	2	
9	2	2	
10	0	1	Priority boosting applied
11	2	1	
12	3	1	
13	0	2	
14	0	2	
15	2	2	
16	2	2	T2 finishes
17	3	2	
18	3	2	
19	0	3	T0 finishes
20	3	1	Priority boosting applied
21	3	2	T3 finishes

   a. When does thread 3 resume execution after being blocked for I/O?

→Thread 3 runs for 1 tick at tick 5, then blocks for I/O. It stays blocked for 4 ticks. After that, it resumes running at tick 12.

b. How does priority boosting after 10 ticks affect the execution order?
→ After 10 ticks, the system boosts all threads that haven't run recently back to the highest priority level. This gives them a better chance to run soon, even if they were in a low-priority queue before.

c. How does MLFQ balance short threads vs. long-running threads?
→ MLFQ gives short threads more chances to run by keeping them in higher priority levels. Long-running threads get moved to lower levels, so they have to wait longer. This helps short threads finish quickly.

2. If a thread repeatedly performs I/O and avoids long CPU bursts, how does MLFQ treat it compared to CPU-bound threads?

→ A thread that does I/O often and doesn’t use much CPU time stays in high priority. It runs more often than CPU-heavy threads, which get moved to lower levels. This makes the system more responsive for I/O tasks.