CS5600 Homework 01
CPU Processes and Scheduling

Hang Zhao, NUID: 002826538

January 22, 2026

Problem Set 1: CPU Scheduling [100 points]

Problem 1 [10 points]

Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs?

The scheduler needs to distinguish between these two types because they have totally different behavior and requirements.

I/O-bound programs spend most of their time waiting for I/O operations (disk reads, network requests, user input). They use short bursts of CPU and then block. If the scheduler doesn't prioritize them, they'll wait forever even though they barely need any CPU time. This kills response time for interactive programs.

CPU-bound programs continuously use the CPU for long periods doing computations. They don't block much and will hog the CPU if given the chance.

If the scheduler treats them the same, CPU-bound jobs will monopolize the CPU while I/O-bound jobs sit in the ready queue even though they could be making progress on their I/O operations. The system ends up with the CPU busy but I/O devices sitting idle, which is inefficient.

By giving I/O-bound programs higher priority or scheduling them first, the scheduler keeps I/O devices busy (they start their I/O and block quickly) while CPU-bound jobs use the CPU during I/O wait times. This improves both response time and overall system utilization.

Problem 2 [30 points]

Discuss how the following pairs of scheduling criteria conflict in certain settings:

(a) CPU utilization and response time

These conflict when you try to maximize both at the same time.

To maximize CPU utilization, you want to run long CPU-bound jobs that keep the CPU busy without context switching overhead. But this means short interactive jobs have to wait in the queue, giving them terrible response times.

For example, if you have a 10-second compilation job and a 50ms text editor keystroke to process, running the compilation first keeps CPU utilization high but the user has to wait 10 seconds to see their keystroke appear (bad response time). If you interrupt the compilation to handle the keystroke, you get good response time but lower CPU utilization because of the context switch overhead.

You can't have both maxed out - you have to pick which one matters more for your workload.

(b) Average turnaround time and maximum waiting time

Optimizing for average turnaround time can make the maximum waiting time really bad for some processes.

If you use Shortest Job First to minimize average turnaround time, short jobs get processed quickly which brings down the average. But long jobs keep getting pushed back and can end up waiting forever if short jobs keep arriving. One long job might wait hours while the average stays low because of all the short jobs finishing fast.

Example: Jobs with times [1, 1, 1, 1, 100] arrive. SJF runs all the 1s first, so average turnaround is good. But the 100-unit job waits through all of them, giving it maximum waiting time of 4 time units before it even starts. If you tried to limit maximum waiting time instead, you'd have to run the long job earlier, which increases average turnaround time.

(c) I/O device utilization and CPU utilization

When a process does I/O, it blocks and the CPU sits idle (or switches to another process). When the CPU is running a process, that process isn't doing I/O.

To maximize CPU utilization, you want processes that never block and just compute constantly. But then your I/O devices are idle because nobody's using them.

To maximize I/O utilization, you want processes constantly doing I/O operations. But while they're waiting for I/O to complete, the CPU has nothing to do.

The only way to get both high is to have a good mix of CPU-bound and I/O-bound processes and schedule them well so I/O-bound processes can use devices while CPU-bound processes use the CPU. But you can't maximize both with a single process - it's either computing or doing I/O, not both simultaneously.

Problem 3 [20 points]

What type of process does the regressive round-robin scheduler favor?

This scheduler favors CPU-bound processes.

Here's why: When a CPU-bound process runs, it uses its entire time quantum without blocking. The scheduler then adds 10ms to its quantum and boosts its priority. So CPU-bound processes get longer time slices and higher priority over time.

When an I/O-bound process runs, it blocks before using its full quantum (to do I/O). The scheduler reduces its quantum by 5ms but keeps the same priority. So I/O-bound processes end up with shorter time slices and don't get priority boosts.

Let me trace through an example:
CPU-bound process A:

- Starts with 50ms quantum, priority P
- Uses full quantum → quantum becomes 60ms, priority increases
- Uses full quantum → quantum becomes 70ms, priority increases again
- After 5 rounds: quantum is 100ms (max), high priority

I/O-bound process B:

- Starts with 50ms quantum, priority P
- Blocks after 20ms → quantum becomes 45ms, priority stays same
- Blocks after 15ms → quantum becomes 40ms, priority stays same
- Keeps shrinking, never gets priority boost

Eventually process A dominates the CPU with its large quantum and high priority, while process B gets squeezed out even though it barely uses the CPU. This is pretty unfair to I/O-bound processes and goes against what most schedulers try to do (favor interactive I/O-bound work).

Problem 4 [40 points]

Which scheduling algorithms could result in starvation?

(a) First-come, first-served - NO starvation

FCFS can't cause starvation. Every process that arrives gets put in the queue and eventually reaches the front. Once you're at the front, you run to completion. Nobody can cut in line.

The wait time might suck if you arrive behind a long job, but you will eventually run. Starvation means never getting to run, which can't happen with FCFS.

(b) Shortest job first - YES, can starve

SJF can definitely cause starvation for long jobs.

If short jobs keep arriving, they always get scheduled before long jobs. A long job could sit in the queue forever if there's a steady stream of short jobs showing up.

Example: Process A needs 100 seconds. But every 10 seconds, a new 5-second job arrives. The scheduler keeps picking the 5-second jobs because they're shorter, and A never runs. That's starvation.

This is why SJF isn't used much in practice for interactive systems - it's too unfair to long processes.

(c) Round robin - NO starvation

Round robin can't starve processes. Every process in the ready queue gets a turn in order. Even if new processes keep arriving, they go to the back of the queue and everyone cycles through.

Each process gets CPU time at regular intervals (at worst, every N * quantum where N is the number of processes). It might get small chunks of time, but it's not starved.

(d) Priority - YES, can starve

Priority scheduling is probably the worst for starvation. Low priority processes can be starved indefinitely if high priority processes keep arriving.

Example: Process A has priority 1 (low), Process B has priority 10 (high). B gets scheduled. Then Process C arrives with priority 10. C gets scheduled. Then Process D arrives with priority 10. And so on. As long as high priority processes keep showing up, A never runs.

This is why real priority schedulers often implement aging (gradually increase priority of waiting processes) to prevent starvation. Without aging, pure priority scheduling is pretty brutal to low priority processes.

Problem Set 2: Processes [50 points]

Problem 5 [10 points]

When a process creates a new process using fork(), which states are shared between parent and child?

Looking at the options:

(a) Stack - NOT shared

The child gets a copy of the parent's stack, but it's not shared. They have independent stacks in different memory spaces. Changes in one don't affect the other.

(b) Heap - NOT shared

Same deal as stack. The child gets a copy of the heap at the time of fork, but after that they're independent. If the parent allocates new memory, the child doesn't see it.

(c) Shared memory segments - YES, shared

This is the only one that's actually shared. Shared memory segments are explicitly created to be shared between processes. After fork, both parent and child have access to the same shared memory regions. Changes made by one process are visible to the other.

So the answer is (c), shared memory segments.

The whole point of fork is that it creates an independent copy of the process, except for things that are explicitly meant to be shared like shared memory segments and open file descriptors.

Problem 6 [10 points]

How many processes are created by the program?

Let me trace through this step by step:

```
int main() {
    fork();  // Line 1
    fork();  // Line 2
    fork();  // Line 3
    return 0;
}
```

Initially: 1 process (the original parent)

After first fork():

- Original process continues
- New child process created
- Total: 2 processes

Both of these processes now execute the second fork().

After second fork():

- Process 1 forks → creates Process 3
- Process 2 forks → creates Process 4

- Total: 4 processes

All four processes now execute the third fork().
After third fork():

- Process 1 forks → creates Process 5
- Process 2 forks → creates Process 6
- Process 3 forks → creates Process 7
- Process 4 forks → creates Process 8
- Total: 8 processes

The pattern is: each fork() doubles the number of processes.

- Start: 1 process
- After fork 1: $1 \times 2 = 2$ processes
- After fork 2: $2 \times 2 = 4$ processes
- After fork 3: $4 \times 2 = 8$ processes

Or using the formula: 2^n where n is the number of forks $= 2^3 = 8$ processes total.
Answer: 8 processes are created (including the initial parent process).

Problem 7 [30 points]

Discuss three major complications that concurrent processing adds to an operating system.

1. Race Conditions and Synchronization

When multiple processes run concurrently, they might access shared resources (memory, files, devices) at the same time. This causes race conditions where the outcome depends on timing.

Example: Two processes try to increment a shared counter. Process A reads value (10), Process B reads value (10), A writes 11, B writes 11. The counter should be 12 but it's 11 because they both read the old value before either wrote back.

The OS needs synchronization mechanisms (locks, semaphores, monitors) to prevent this. This adds complexity - the OS has to manage these primitives, detect deadlocks, handle priority inversion, etc. Without concurrent processing, you don't need any of this.

2. Context Switching Overhead

With concurrent processing, the OS constantly switches between processes to give the illusion they're all running simultaneously. Each context switch requires:

- Saving the current process's registers, program counter, stack pointer
- Loading the next process's state
- Flushing and reloading caches and TLBs
- Updating scheduling data structures

This takes time and doesn't do any useful work - it's pure overhead. The OS has to balance between switching often (good responsiveness) and switching less (lower overhead). Without concurrency, you just run one program start to finish with no switching.

3. Resource Management Complexity

With multiple processes, the OS has to carefully manage limited resources:

- Which process gets the CPU next? (scheduling algorithms)
- How much memory does each process get? (memory allocation)
- What if there's not enough memory? (paging, swapping)
- Who gets to use the printer? (resource allocation)
- What if two processes want the same file? (file locking)

The OS needs complex algorithms to handle all this fairly and efficiently. It has to prevent deadlock (Process A waits for resource held by B, B waits for resource held by A). It has to decide which processes to kill if resources run out. It has to track what each process owns.

Without concurrent processing, resource management is trivial - there's only one process, so give it everything.