Technical University of Denmark

02159 Operating Systems
DTU Compute
Fall 2025

Exam Answers
Operating Systems

December 11, 2025 Vincent Van Schependom (s251739)

1 Short questions

Question 1: What is the key advantage of microkernel operating systems compared to monolithic operating systems? Explain briefly your answer. **(4 points)**

ANSWER

Microkernel operating systems are **more *stable* and more *flexible*** than monolithic systems.

In microkernel systems, the kernel is kept very small: only basic services are run in kernel mode. All other OS services are run in user mode. This ensures *stability*, which is not guaranteed in monolithic systems: in monolithic systems, the whole operating system runs in kernel mode, which means that a bug in for example a driver can cause the entire system to crash.

Furthermore, an extension of a monolithic system requires the whole kernel to be rebuilt, while an extension of a microkernel system only requires the relevant modules to be rebuilt. This makes microkernel systems more flexible than monolithic systems.

Question 2: You are developing an application that is composed of multiple threads. Each thread updates a global variable. Which method you would use to avoid a race condition? Explain briefly your answer. **(4 points)**

ANSWER

I would use a *mutex* to ensure mutual exclusion.

Mutexes are specifically designed to protect shared variables (like global variables or heap memory) among threads within the same process. These threads share the same address space (of their process), so they can access the same global variables.

By acquiring the mutex before updating the variable and releasing it afterward, we ensure that *only one* thread accesses the critical region at a time, preventing race conditions.

Question 3: Provide a real-world example of an application where the NFU (Not Frequently Used) page replacement algorithm will perform very poorly. Do not use the examples provided in the textbook or slides. Explain briefly your answer. **(4 points)**

ANSWER

NFU doesn't forget: if a page is used a lot in the beginning of the process, but afterwards never again, the use frequency will be high and it will thus never be evicted. This implies that there is less space for pages that are used later on a (less) frequent basis.

That's why it is for example a bad idea to use NFU for a **web browser**. When opening a web page, many resources (HTML, CSS, JavaScript, images, etc.) are loaded into memory. These resources are used frequently during the initial loading phase of the page, but once the page is fully loaded, many of these resources are not used anymore.

Another example is a **text editor** that opens a large file. The initial part of the file is accessed frequently while loading and displaying it, but once the user starts editing or scrolling to other parts of the file, the initial pages become less relevant.

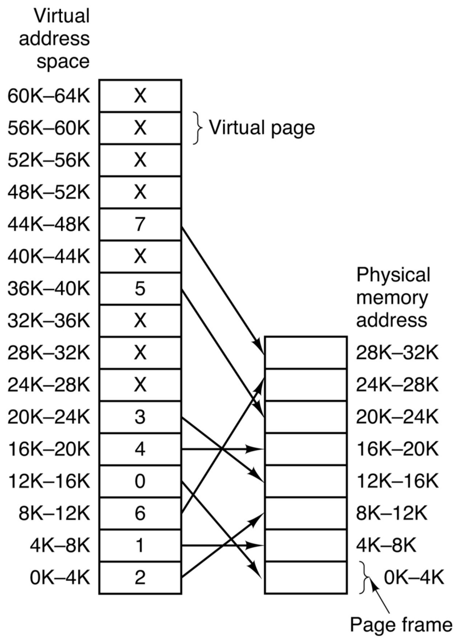

Figure 1

Question 4: A program is executed and the current state of the memory is as shown in Figure 1. The next two instructions add two integers: the first integer is in memory address 50 000 and the second integer is in memory 50 004. How many page faults are raised? Explain briefly your answer. For simplicity, you can assume that the addresses shown in the gure are in thousands: for example 4K-8K is equal to 4000-8000. **(4 points)**

ANSWER

Only *one* page fault will be raised.

The virtual page from virtual address 48 000 to 52 000 is not mapped to any physical frame (X in the figure). Hence, when getting the first integer from virtual address 50 000, a page fault will be raised and the physical page frame corresponding to the virtual page [48 000 - 52 000) will be loaded into memory by the MMU, which uses the page table for this. *(Note that a page replacement takes place, i.e. a physical frame is evicted to make place for the new page.)*

When the second integer is accessed from virtual address 50 004, no page fault will be raised, since the virtual page [48 000 - 52 000) is now mapped to a physical frame. In other words, the physical frame is now loaded in main memory (RAM) and can be accessed directly.

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

int counter = 1;

void child(void) {
    counter++;
    exit(0);
}

int main(void)
{
  if (fork() == 0) {
    child();
  }
  waitpid(-1, NULL, 0);

  counter++;
  printf("%d\n", counter);
  return 0;
}
```

Figure 2

Question 5: What will the code in Figure 2 print? Explain briefly your answer. You can assume that all system call invocations are successful. **(4 points)**

ANSWER

The code will print `2`**.**

Only the child process wil run the `if`-statement, since `fork()` returns 0 in the child process and a positive value in the parent process. The child and parent processes have different physical address spaces, so the child will increment its *own* copy of `counter` from 1 to 2, after which it `exit()`s. The parent process waits for any child process to exit (`waitpid(-1,...)`), after which it increments its own copy of `counter`, being 1. Finally, it prints the incremented value of `counter`, being $1 + 1 = 2$.

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

pthread_t thread_id[5];

void* count(void *a) {
  printf("%d\n", *(int*)a);
  pthread_exit(NULL);
}

int main(void) {
  int i;
  for (i=1;i<=5;i++) {
    pthread_create(&thread_id[i], NULL, count, &i);
    pthread_join(thread_id[i], NULL);
  }
  return 0;
}
```

Figure 3

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

int A = 0;
pthread_t thread_id[1000];

void* count(void *input) {
  int i;
  for (i=0;i<1000;i++)
    A++;
  pthread_exit(NULL);
}

int main(void) {
  int i;
  for (i=0;i<1000;i++)
    pthread_create(&thread_id[i], NULL, count, NULL);
  for (i=0;i<1000;i++)
    pthread_join(thread_id[i], NULL);
  printf("%d\n", A);
  return 0;
}
```

Figure 4

Question 6: What will the code in Figure 3 print? If the code is executed multiple times, will the printed numbers always be in the same order? Explain briefly your answer. You can assume that all system call invocations are successful. **(4 points)**

ANSWER

The code will print the numbers 1, 2, 3, 4, and 5 on separate lines. If the code is executed multiple times, **the printed numbers will always be in the *same* order**.

The `pthread_join` function is called *inside* the loop, immediately after each thread is created. This join blocks the calling process (the `main` thread) and forces it to wait for the specific thread to exit before continuing execution. This ensures **sequential** execution: the main thread cannot increment the loop variable `i` or create the next thread until the current thread has finished printing and exited.

In this case, there is *no* race condition on the shared variable `i`, and the threads run strictly one after another in the order of the loop.

Question 7: What will the code in Figure 4 most likely print? Explain briefly your answer. You can assume that all system call invocations are successful. **(4 points)**

ANSWER

The code will most likely print **a value *less than*** 1 000 000.

Each of the 1000 threads will try to increment the shared (process-wide) variable `A` 1000 times. However, `A++` is *not* an atomic operation: it first loads the value of `A` into a register, increments it, and then stores it back into `A`. There is thus a **race condition** between the threads: a thread can get blocked after loading the value of `A` into a register. While it is blocked, other threads may read, increment, and update `A`. When the blocked thread resumes, it increments its old value of `A` – which was stored in the thread-local register – and writes it back to `A`, effectively overwriting the progress made by the other threads. This results in “lost updates”, causing the final value of `A` to be lower than the expected 1 000 000.

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

int main(void){
  fork();
  fork();
  printf("a\n");
  exit(0);
}
```

Figure 5

Question 8: How many times will the letter ‘a’ be printed if we execute the code in Figure 5? Explain briefly your answer. You can assume that all system call invocations are successful. **(4 points)**

ANSWER

The letter ‘a’ will be printed **4 times**.

The parent (original) process will create a new child on the first `fork()`. Both processes (parent and child) will execute all code below the first `fork()`. This means that both parent and child will create a new child on the second `fork()`. In total, we now have 4 processes (the original parent, the original child and the new two children of each of them). Each of these processes will execute the `printf("a")` statement, resulting in 4 prints of the letter ‘a’.

Question 9: A typical example of a deadlock appears when making transactions on multiple accounts protected by multiple mutexes, i.e., *each account* is protected by a mutex. One solution that is safe from race conditions and deadlocks is having *one single* mutex over *all* the accounts. What is the disadvantage of this solution? Explain briefly your answer. **(4 points)**

ANSWER

There will be great **lock contention**, causing a significant **performance degradation** because of reduced concurrency.

With a single mutex protecting all accounts, only one thread can access *any* account at a time. This means that if multiple threads want to perform transactions on different accounts simultaneously, they will have to wait for the mutex to be released, even if they are working on completely independent accounts. This effectively makes the whole system *sequential*, throwing away the benefits of multithreading and (pseudo)parallelism.

Question 10: Is Direct Memory Access (DMA) better to be used when the operating system is busy serving a large number of *I/O-bound* processes or a large number of *CPU-bound* processes? Explain briefly your answer. **(4 points)**

ANSWER

DMA is more beneficial if the operating system is busy serving a large number of **CPU-bound** processes, rather than I/O-bound processes.

CPU-bound processes spend most of their time performing computations and using the CPU (long bursts of CPU activity). If these processes need to read/write data from/to I/O devices, using DMA allows the CPU to continue executing other tasks while the DMA controller handles the data transfer in the background. I/O-bound processes only have short bursts of CPU activity followed by long periods of waiting for I/O operations to complete. During these waiting periods, the CPU is already idle, and thus DMA provides less of an advantage, because there is overhead involved for setting up the DMA transfer: configuring the DMA controller, initiating the transfer, and handling interrupts.

(*Offloading to DMA improves overall system throughput compared to* Programmed I/O, *where the CPU continuously polls the device (busy-waiting) until the transfer is complete, and precious compute-time would thus be wasted. Compared to* Interrupt-Driven I/O, *on the other hand, it reduces the number of interrupts from one per character to one per buffer.*)

Question 11: Describe an OS functionality that is important for user experience to be implemented using interrupt-based I/O. Justify briefly your answer. **(4 points)**

ANSWER

An important OS functionality that should be implemented using interrupt-based I/O is **handling the input of peripherals**, where a short response time is crucial for a good user experience.

Let's say we're creating a **keyboard driver**. When we press a key, the keyboard sends an interrupt to the CPU, which **immediately** handles the key press event. This makes the system more **responsive** to user input. If we would implement this without interrupt-driven I/O, we would have to periodically check the keyboard status, which is *inefficient* and can lead to *delays*.

Question 12: A program needs to find something in a very large array of data. The parent process divides the search space in four pieces and spawns *four child processes.* Each child process searches one piece of the search space in parallel. The child process that finds it reports back to the parent process. Everything mentioned up to here is already implemented.

Your task is to implement the final part that follows: The *parent* process needs to stop the other three *child* processes that are still searching. Which interprocess communication method you would use to implement this functionality? Which system call corresponds to it? Briefly explain your answer.
(4 points)

ANSWER

I would use **signals** (via `kill()`) for interprocess communication to stop the other three child processes, once the parent process sees (via `waitpid()`) that one child process has exited after finding the target.

When the parent creates the 4 child processes, it stores their PIDs using `int pids[4];` and then in a loop for $i = 1, \ldots, 4$, it assigns `pids[i] = fork();`. It then listens for *any* child process to exit using `waitpid(-1, &answer, 0)`. When a child j finds the answer in its search space, it exits using `exit(answer);`. The parent process then knows that one child has found the answer, and it can send signals (like `SIGTERM`) to the other three child processes using `kill(SIGTERM, pids[i]);` for $i \in \{1, \ldots, 4\} \setminus \{j\}$,

Question 13: In which circumstances mutual exclusion with a spinlock is good for efficiency? Explain briefly your answer. **(4 points)**

ANSWER

Spinlocks are efficient in multiprocessor systems **when the lock** (e.g. a mutex) **is held for a *very short time*** (e.g. shorter than the time it takes to perform a context switch).

Spinlocks use busy waiting, which wastes CPU cycles. However, putting a thread/process to sleep (blocking) involves a context switch, which involves *overhead* and *invalidating* the CPU *cache*. If the wait time is shorter than the time it takes to perform a context switch, spinning may be faster than blocking.

Question 14: What is the main advantage of preemptive scheduling compared to nonpreemptive scheduling? Explain briefly your answer. **(4 points)**

ANSWER

Preemption allows the operating system to interrupt a process at any point, blocking it and allowing another process to run. This means that the operating system can **prevent a process from running for too long**, ensuring better responsiveness and fairness. In a non-preemptive system, a process must explicitly yield the CPU to another process, which can lead to a process **monopolizing** the CPU.

Question 15: What is the main advantage of multiprocessor systems compared to single-processor systems? Explain briefly your answer. **(4 points)**

ANSWER

True parallelism.

Single-processor systems achieve *pseudo*-parallelism by rapidly switching between threads (multi-threading). Multiprocessor systems can execute multiple instructions from different threads or processes at the exact same physical time on different CPUs, offering *true* parallelism and increased system throughput.

Question 16: Why communication protocols are necessary for implementing distributed applications over the Internet. Explain briefly your answer. **(4 points)**

ANSWER

A distributed system consists of loosely coupled computer systems that might run different *operating systems*, different *hardware architectures*, and belong to different *organizations*. This makes *coordination protocols* essential. These protocols define a set of rules and conventions (a **common language**) for data exchange. They ensure that data is formatted, transmitted, and interpreted correctly by both the sender and receiver, enabling **interoperability between diverse systems and applications** over the Internet.

2 Long questions

Question 1 (12 points)

Let's assume that you are designing a server for the OS Challenge. Everything is as speciﬁed on the OS Challenge speciﬁcation document with one important difference: the incoming requests have *variable difficulty* (i.e., the difference between the start and the end is not constant). How would you design the scheduler? Motivate your answer.

ANSWER

Since the score ($\texttt{score} = \frac{1}{\texttt{total}} \sum \texttt{delay} \cdot \texttt{priority}$) does not *explicit* distinguish between difficult and easy requests, the only thing that matters for difficult requests is that they are handled *quickly*, to minimize the `delay`. Furthermore, easy tasks should not be negatively impacted by trying to optimize for difficult tasks.

I would solve the difficulty imbalance by **splitting up requests into smaller sized "chunks"** of a fixed `CHUNK_SIZE` (e.g., 10 000 numbers in the search space per chunk). This way, every request, regardless of its initial difficulty, will be split up into tasks (disjoint parts of the search space) that are – on average – of equal difficulty. This distributes the work evenly across threads, preventing situations where one thread is stuck processing a very difficult request while others are idle.

There are 4 CPU cores available on OS Challenge system, so I would use 1 dispatcher thread and **4 worker threads**. The dispatcher is *I/O-bound*: Its job is to `accept()` a connection and read the request. For the vast majority of time, this thread is blocked (sleeping) in the OS kernel, waiting for an incoming packet. The workers are *CPU-bound*: their job is to compute SHA-256 hashes. This is purely mathematical and will utilize 100% of a CPU core's cycles without blocking for I/O.

When a TCP request arrives, the OS (**scheduler**) wakes up the Dispatcher thread. It momentarily preempts one of the Worker threads on one core. The Dispatcher runs for a tiny fraction of a second (to accept and queue), then goes back to sleep. The preempted Worker immediately resumes. In this way, we utilize nearly 100% of the computing power, instead of ∼75% if we would use 3 worker threads instead of 4. The overhead of the context switch for the Dispatcher is negligible compared to the massive throughput gain of having that 4th core crunching hashes 99% of the time.

The dispatcher thread does **not** handle *all* chunking *upfront*. Instead, it pushes the *entire* request (possibly very large) onto the local **double-ended** queue of a worker thread. The worker thread then splits the request into chunks *lazily*: it pops a new requests from the *head* of the local queue. If the task range exceeds `CHUNK_SIZE`, the *worker* splits it. It processes the first chunk immediately and pushes the remaining large range back onto the *head* of its local queue.

I would also allow **work stealing** between the worker threads to further balance the load. A thread that is idle can acquire the lock on the *tail* of another worker's local queue and steal a chunk of work from it. Why the tail? Well, this part of the double-ended queue contains the largest (and coldest) tasks. Furthermore, the thread that owns the queue is working on the head, so stealing from the tail minimizes contention.

If a thread finds the answer in a chunk, the other chunks belonging to that request can then be ignored. This is done by keeping track of active requests in a global hash table. When a thread pops a chunk, it first checks if the request is still active; if not, it discards the chunk.

NOTE: this answer is very similar to the proposed experiment in Long Question 3. Crucially, however, in my answer later on, I also discuss the Hypothesis and how I would test it. Furthermore, I discuss the Motivation based on our previous experiments.

Question 2 (12 points)

i. Summarise the advantages and disadvantages of processes and threads.

ii. You are developing a Domain Name System (DNS) server. When a DNS server receives a request from a client, it first looks if this domain name is in a local cache, otherwise it makes a request to a higher-tier DNS server. Once it retrieves the IP address, it responds to the client. The DNS server should handle multiple requests from potentially hundreds of clients in parallel. Would you implement the DNS server using multiple processes or multiple threads? Motivate your answer and discuss if the disadvantages of your solution are relevant in this use case. **If relevant**, also discuss how you would overcome them.

ANSWER

i. Advantages and disadvantages of threads and processes:

- **Processes:** The primary advantage is *isolation*; a bug or crash in one process does not affect others, and they are secure from one another due to separate address spaces. They are also easier to use and implement than threads (no synchronization needed). The disadvantages include high resource *overhead* (memory, CPU) for creation and destruction, expensive context switching, and the need for Inter-Process Communication (*IPC*) mechanisms (like shared memory, pipes, etc.) to share data.

- **Threads:** The advantages are that they are *lightweight*; creation, destruction, and switching are much faster than for processes. They also share the same address space, making data sharing and *communication* very efficient. The disadvantages are the *lack of protection* (one crashing thread can crash the entire process) and the complexity of *synchronization* to prevent race conditions when accessing shared resources.

ii. I would implement the DNS server using **multiple threads**.

- **Shared state (cache):** Threads within the same process share the same address space and global variables. This makes accessing and updating the shared DNS cache extremely efficient and straightforward. If we used processes, which have distinct address spaces, sharing the cache would require complex Inter-Process Communication (IPC) mechanisms.

- **Performance:** Threads are more lightweight than processes. Creating and destroying threads is much faster than creating processes via `fork()`. For a high-performance server handling hundreds of clients, the overhead of process switching would be significant, whereas thread switching is faster.

Disadvantages of threads and their respective solutions:

- **Synchronization:** A major disadvantage of threads is that the OS provides no protection on shared resources, leading to potential race conditions. This is highly **relevant** here because multiple threads will try to read from and write to the shared DNS cache simultaneously.

  Solution: This must be overcome by using a synchronization technique, such as **mutexes**, to ensure mutual exclusion when accessing the cache.

- **Stability/isolation:** Threads lack isolation; a bug (e.g., a segmentation fault) in one thread can crash the entire process, stopping the service for all clients. This, however, is **not really relevant** in the context of a DNS server, as translating domain names to IP addresses is a simple and stable operation.

  *(*Solution: *One could implement a* watchdog timer *to detect if the process hangs or crashes and automatically restart the process, minimizing downtime. However, this decreases performance and hence there is a trade-off between stability and performance.)*

Question 3 (12 points)

After the presentation of the experiments of all other groups, propose a new experiment for the OS Challenge that your group has not tested before. Motivate the experiment by explaining why you believe it will improve the performance and describe how you would test if your hypothesis is true.

ANSWER

Proposed experiment:

I would design a **dispatcher-worker** model with **voluntary yielding** (manual preemption) and a **hybrid FIFO queueing** system. *NOTE: the chunking was already covered in Long Question 1*

Motivation:

The scoring formula requires processing high-priority requests as fast as possible. My teammates' previous experiments highlighted two critical bottlenecks:

1. **Experiment 5** showed that relying on *standard OS Scheduling* (e.g., using separate high-priority threads) introduces *significant overhead* because the OS must force a context switch every time a high-priority task arrives, outweighing the benefits of reordering.
2. **Experiment 4b** showed that "chunking" increases throughput, but performing the split in the main thread (creating *all* chunks of a request *up front*) causes a dispatcher bottleneck: heavy global queue contention and starvation of other threads.

Hypothesis:

We can minimize the weighted latency `score` by implementing **voluntary yielding** and **lazy chunking**. By using simple FIFO queues ($O(1)$) instead of a max-heap priority queue ($O(\log n)$) and allowing worker threads to *voluntarily* yield to high-priority tasks, we avoid the overhead of the *extra* context switches that occur when waking up distinct high-priority threads. Finally, by splitting tasks *lazily* in the worker threads rather than the main thread all at once, we eliminate the dispatcher bottleneck.

Implementation

One thread (the listener) is dedicated to listening for new requests and enqueuing them in the appropriate queue based on priority level p: if $p > 1$, it enqueues the request in the `Global_High_Queue`, otherwise (if $p = 1$) in the **double-ended** `Local_Low_Queue` of the *next* worker thread (load balancing based on `next_worker_id = (next_worker_id+1) % num_workers`).

There are 4 CPU cores available, so I would use 1 dispatcher thread and **4 worker threads**. The dispatcher is *I/O-bound*: Its job is to `accept()` a connection and read 49 bytes. This takes microseconds. For the vast majority of time, this thread is blocked (sleeping) in the OS kernel, waiting for an incoming packet. The workers are *CPU-bound*: their job is to compute SHA-256 hashes. This is purely mathematical and will utilize 100% of a CPU core's cycles without blocking for I/O. When a TCP request arrives, the OS wakes up the Dispatcher thread. It momentarily preempts (pauses) one of the Worker threads on one core. The Dispatcher runs for a tiny fraction of a second (to accept and queue), then goes back to sleep. The preempted Worker immediately resumes. In this way, we utilize nearly 100% of the computing power, instead of ~75% if we would use 3 worker threads instead of 4. The overhead of the context switch for the Dispatcher is negligible compared to the massive throughput gain of having that 4th core crunching hashes 99% of the time.

The priority levels of requests are generated using an exponential distribution. This distribution means that the vast majority of requests will have the lowest priority level ($p = 1$), while high-priority requests ($p > 1$) will appear rarely 'from time to time'. Therefore, an efficient threshold is likely to split the queues into Priority 1 ("bulk" traffic) and Priority > 1 ("urgent" traffic).

Crucially, the main thread does NOT split the request (like is done in the current implementation) into chunks *all at once*, as already explained in Long Question 1. If the main thread tried to split a range of 1 000 000 hashes into 1000 chunks up front, it would be stuck in a loop (blocking new connections) and would flood the memory with task structs. Instead, the splitting happens *lazily* (when needed) by the worker threads.

Each worker runs a loop that performs **voluntary yielding**:

1. **Check urgent (lock-free):** Perform an atomic read on the `Global_High_Queue` size. This is a cheap operation. Only if the size > 0 does the thread acquire the mutex to process the high-priority task immediately.
2. **Local work:** If no high-priority work exists, pop a task from the `head` of the thread's own `Local_Low_Queue`. Next, check if the request is still active (not already answered by another thread) using a quick lookup. If the request is not active anymore, discard and restart the loop.
3. **Lazy chunking:** If the task range exceeds `CHUNK_SIZE`, the *worker* splits it. It processes the first chunk immediately and pushes the remaining large range back onto the `head` of its local queue. This prevents the "freezing" seen in Exp 4b because the main thread never loops; the worker generates work as needed.
4. **Work stealing:** If the local queue is empty, try to steal a task from the `tail` of another worker's queue (balancing the load). Acquire the specific mutex to steal work from the tail (FIFO order) of that threads `Local_Low_Queue`, taking the largest (and coldest) available chunks without disturbing the owner's work at the head.
5. **Yield:** After processing one small chunk, the loop restarts at Step 1. This allows a high-priority request to "interrupt" a low-priority task within milliseconds without requiring the OS to perform a context switch to wake up a different thread.

Testing:

We will run the `run-client-highspread-priority-short-delay.sh` script. This workload (high concurrency, mixed priorities) caused previous implementations to fail due to either blocking (single global queue and up front chunking) or overhead (OS-based priority scheduling). We expect this hybrid solution to achieve the lowest weighted latency score by combining the low overhead of FIFO with the responsiveness of voluntary preemption.