Technical
University of
Denmark

02159 Operating Systems
DTU Compute
Fall 2025

Exam Questions

Operating Systems

Fall 2025

Vincent Van Schependom

Introduction

This document was created by me, Vincent, as a preparation for the exam of the course Operating Systems at DTU. It contains questions from the exams in 2021, 2022, and 2024. All answers were written out by myself and are not guaranteed to be correct, although they are based on the course material (slides and coding examples) covered in class.

Exam Information

Aids allowed All aids

Exam duration 4 hours

Weighting According to their respective points (100 points in total)

Further notes:

- All questions contribute toward the final grade. The maximum number of points awarded by each correctly answered question is listed next to the question.
- For short questions, the following apply: (i) a brief answer is expected; (ii) you are also expected to briefly explain your answer in approximately 2-3 lines; (iii) a correct answer with a correct explanation yields 4 points in total.
- For long questions, the following apply: (i) there might be more than one correct answer; (ii) you are expected to briefly explain your answer in approximately half a page; (iii) a complete and correct answer yields 12 points; (iv) if you make additional assumptions, remember to briefly explain them.

Contents

1 Short questions

1.1 General questions

Question 1: Name a communication method that can be used between threads of the same process but cannot be used for communication between parent and child process. Explain briefly your answer. **(4 points)**

ANSWER

Different threads inside a process share the (physical) address space of that process. They can thus have direct access to *global variables* and *heap memory* and communicate in these ways.

However, since each process has its own physical address space, threads of *different* processes do *not* share memory by default. This also applies when we call `fork()`: the child process is an exact copy of the parent, but with a different physical address space. While shared memory can be set up for processes, it requires system calls (e.g. `mmap()`). Threads have this implicitly.

Question 2: 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. 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 3: The DMA chip can transfer data from and to the memory without using the CPU. Name a scenario that it is not efficient to use DMA for memory transfer? Explain briefly your answer. **(4 points)**

ANSWER

A scenario where it is not efficient to use DMA for memory transfer is when transferring **very small amounts of data**, such as a few bytes or a single word. This because the overhead of setting up the DMA transfer (configuring the DMA controller, initiating the transfer, and handling interrupts) may outweigh the benefits of offloading the transfer from the CPU. In such cases, it may be more efficient for the CPU to handle the transfer directly, especially if the data transfer is quick and does not significantly impact CPU performance.

Furthermore, in (non-battery-powered) embedded systems, getting rid of DMA makes sense, as it saves money. (It is, however, more energy-efficient in battery-powered embedded systems.)

Question 4: Name a scenario that it is efficient to use DMA for memory transfer? Explain briefly your answer. **(4 points)**

ANSWER

It is efficient when the CPU is busy with other tasks (high CPU utilization) and the data transfer is not super small.

By offloading the data transfer to the DMA, the CPU is free to execute other processes or threads in parallel while the transfer takes place. The DMA issues an interrupt when it's done transferring all the data in its buffer. This improves overall system throughput compared to *Programmed I/O*, where the CPU continuously polls the device (busy-waiting) until the transfer is complete. Compared to *Interrupt-Driven I/O*, on the other hand, it reduces the number of interrupts from one per character to one per buffer.

Additionally, it is efficient in battery-powered embedded systems, as the DMA chip is more energy-efficient compared to moving the data through the CPU.

Question 5: Which is the purpose of a watchdog timer? Explain briefly your answer. **(4 points)**

ANSWER

A watchdog timer is a timer that resets the system if it expires.

It gets reset by the OS periodically, which means that if it expires, the OS is frozen (e.g. because of an infinite loop or a deadlock in the kernel). In this case, the watchdog timer will reset the system, such that the OS can run again.

Question 6: Describe a bug or error that would trigger a watchdog timer reboot? Explain briefly your answer. **(4 points)**

ANSWER

An infinite loop or a deadlock in the OS kernel would trigger a watchdog timer reboot.

A watchdog timer is a hardware or software timer that reboots the system if it expires. That is, it is used to recover from the unresponsive state.

Normally, the watchdog timer is reset periodically by the OS. However, if a kernel bug or error causes the OS to freeze (e.g. an infinite loop or a deadlock), the watchdog timer is not reset anymore, and it will eventually expire and trigger a reboot.

Infinite loops in *programs* do *not* cause a watchdog timer reboot if we use a preemptive scheduler. In that case, the OS can interrupt the application after its quotum of CPU time expires and it can then switch to another process.

Question 7: Which is the key advantage of developing device drivers using interrupt-driven I/O? Explain briefly your answer. **(4 points)**

ANSWER

The key advantage is that it eliminates *busy waiting* (or polling). It is (i) more efficient and (ii) more responsive.

An interrupt-driven I/O is an I/O method where the CPU is notified by the device when it is ready to transfer data, rather than the CPU continuously checking the device status (polling).

(1) The OS can put the calling process to sleep and switch to executing other processes while the I/O device performs its task. You don't waste the CPU with busy-waiting. The CPU can thus perform other tasks while waiting for the I/O operation to complete.

(2) More responsive: 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 8: In computer systems with multiple processors, name a challenge that characterises distributed systems as opposed to multicore processors? Explain briefly your answer. **(4 points)**

ANSWER

A major challenge is the lack of shared memory and the reliance on network communication.

In multicore systems (multiprocessors), processors share main memory and can communicate very quickly (nanoseconds). In contrast, distributed systems consist of independent computers that must communicate via message passing over a network, which introduces significantly higher latency (milliseconds) and unreliability issues, such as lost or out-of-order packets.

Furthermore, unlike a multicore system which runs a single shared OS, a distributed system consists of independent nodes that may run different operating systems and belong to different organizations. This makes *coordination* (via protocols) and *load balancing* much harder than just managing threads on a single OS.

Question 9: Name four (4) methods that can be used for communication between two processes that run on the same computer? Explain briefly your answer. **(4 points)**

ANSWER

1. **Shared memory** (via `mmap`): Two or more processes share a specific region of memory (or the OS kernel memory). It is fast but requires careful synchronization (e.g. via mutexes).
2. **Files** (via `open`): Processes communicate by reading and writing data to the same file on the disk. This method requires explicit locking mechanisms (like `lockf`) to avoid race conditions when multiple processes access the file simultaneously.
3. **Pipes** (via `pipe`): A unidirectional data channel managed by the kernel that connects the standard output of one process to the standard input of another.
4. Sockets (via `socket`): An endpoint for communication that allows processes to exchange data streams or messages. While often used for networking (TCP/UDP), UNIX domain sockets are used for efficient bidirectional IPC on the same machine.
5. Via the status code of a child process (`wait` / `waitpid`): A parent process pauses its execution to wait for a child process to terminate. The operating system passes the child's exit status (an integer) back to the parent, allowing the child to communicate its final state or result.
6. **Signals** (via `kill`): Signals are asynchronous software interrupts sent to a process to notify it of an event (e.g., `SIGALRM`). They interrupt the normal flow of execution to run a specific handler function, acting as a control mechanism rather than a data transfer channel.

Question 10: Assuming the operating system uses priority-based scheduling, I/O bound processes should be treated as high or low priority? Explain briefly your answer. **(4 points)**

ANSWER

I/O bound processes should be treated as *high priority.*

I/O bound processes spend most of their time waiting for I/O operations to complete and require the CPU only for *short bursts.* Assigning them high priority ensures that when they become ready, they can quickly acquire the CPU to process data or initiate the next I/O request. This minimizes response time (users experience less latency) and keeps I/O devices busy, whereas a low priority would cause them to wait behind CPU-bound processes, leaving peripherals idle, because these CPU-bound processes use the CPU for longer periods.

Question 11: What is the key advantage of monolithic operating systems compared to microkernel operating systems? Explain briefly your answer. **(4 points)**

ANSWER

In monolithic systems, the whole operating system runs in kernel mode, which is faster and more efficient than microkernel systems, where only the core fundamental services run in kernel mode, while other services run in user mode.

User mode is *costly*: to access protected resources, user mode services must use system calls to make a transition to kernel mode. These transitions between kernel mode and user mode are expensive and slow down performance. That's exactly why microkernel systems are less efficient than monolithic systems, even though they are more stable and more flexible than the latter.

When choosing which parts to run in kernel mode and which to run in user mode, we thus make a trade-off between stability and efficiency.

Question 12: 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.

(The user-mode services (like a file server or device driver) need to talk to each other to function. However, because they are separate processes in user mode, they cannot directly access each other's memory. To communicate, a service sends a message to the kernel (via a system call). The kernel then passes that message to the intended recipient service.)

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 13: Which POSIX system call should you use to send the `SIGUSR1` signal to a child process? Explain briefly your answer. **(4 points)**

ANSWER

Using `kill(child_pid, SIGUSR1)`, we can send the `SIGUSR1` signal to a child process.

The `kill()` system call takes two arguments: the process ID of the process to send the signal to, and the signal to send. In this case, we want to send the `SIGUSR1` signal to a child process, so we can use `kill(child_pid, SIGUSR1)`, assuming that the child process id was stored as follows: `int child_pid = fork();`

Question 14: What is the primary role of the Memory Management Unit (MMU)? Explain briefly your answer. **(4 points)**

ANSWER

The primary role of the Memory Management Unit (MMU) is to manage virtual memory.

More specifically, it maps *virtual pages* to *physical frames* and enforces *logic* (e.g. protection, presentness, ...) along the way. It does this using a *page table* (often organized as a *multi-level* hierarchy to save memory).

During this process, the MMU enforces logic via *control bits* in the *page table entry*:

- **Present Bit:** A page fault occurs if the page is not in RAM.
- **Protection Bits:** Read/Write/Execute permissions.
- **Referenced Bit:** Used by page replacement algorithms.
- **Changed Bit:** If page was modified, it requires writing back to disk before eviction.

Virtual pages are "chunks" of the virtual address space of the process. They allow us to run programs that are much bigger than the available physical memory, by swapping parts of the program in and out of RAM as needed.

Question 15: Explain the challenges associated with static relocation and why it is necessary to transition to dynamic relocation. Discuss the advantages of dynamic relocation in overcoming these challenges. **(4 points)**

ANSWER

Static relocation involves assigning fixed memory addresses to programs at compile time. It's not flexible and doesn't allow for moving processes into memory at runtime.

Static relocation is needed when running multiple programmes without memory abstraction (i.e. without virtual memory). In this case, each programme occupies one 'block' of the physical memory and we need to *add the base address* of these blocks to the *memory references* of instructions in those programmes. This is problematic because it requires modifying the executable code upon compilation and makes it difficult to move processes in memory during runtime.

Using dynamic relocation, the OS doesn't need to change memory addresses in the programs. Instead, the address in the *base register* is added to every memory reference *at runtime* by the CPU. Simultaneously, a *limit register* is checked to ensure the process does not access memory outside its allocated block. This hardware-based approach solves both the addressing problem and the protection problem (replacing memory protection keys, like in static relocation) efficiently.

Question 16: Name a system operation that is not possible without clock interrupts. Explain briefly your answer. **(4 points)**

ANSWER

Preemptive Scheduling (and signaling)

A preemptive scheduler allows a process to run for a specific time slice (quantum). It relies on a hardware clock to issue an interrupt at the end of this interval to force the running process to stop and return control to the scheduler. Without clock interrupts, the operating system cannot force a process to yield the CPU, making preemptive scheduling impossible.

An alarm signal, sent with `signal(SIGALRM, handler)` and `alarm(seconds)`, also requires clock interrupts to be able to signal the process when the alarm expires.

Question 17: Describe a scenario where interrupt-based software is resource efficient. Justify briefly your answer. **(4 points)**

ANSWER

A process waiting for a slow I/O operation, such as a printer finishing printing a character or waiting for a key press from a keyboard.

In a “programmed I/O” (busy waiting) approach, the CPU continuously polls the device status, wasting CPU cycles and energy. With interrupt-based software, the CPU puts the waiting process to sleep and switches to other tasks. The device controller issues an interrupt only when it is ready, allowing the CPU to be used for other useful work in the meantime.

Question 18: 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 (multithreading). 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 19: When is implementing mutual exclusion with a spinlock 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 20: Nodes in a distributed system typically use protocols that have been standardised by international standardisation bodies to communicate with each other. Are international standards also necessary when nodes in a multicomputer system communicate with each other? Explain briefly your answer. **(4 points)**

ANSWER

The agreement does *not* need to be internationally standardised, as long as all nodes in the multicomputer system agree on the same protocol.

Unlike distributed systems, which often span the globe and involve many organizations, multicomputers are typically located in the same room or rack and administered by a single organization (like in a Local Area Network). They use a *dedicated interconnect* rather than a traditional public network.

Therefore, they can use specialized or proprietary protocols optimized for high speed and low latency on that specific hardware, as broad interoperability with the outside world is not required internally. However, developing such protocols is time-consuming and requires specialized knowledge, which is why many multicomputers use existing protocols.

Question 21: Consider preemptive and non-preemptive scheduling. Identify two real-life application-s/senarios where:

i. Preemptive scheduling is better suited than non-preemptive scheduling.

ii. Non-preemptive scheduling is more appropriate than preemptive scheduling.

For each example, explain briefly why you chose the particular scheduling technique and how it benefits the application. **(4 points)**

ANSWER

i. **Interactive User Interface (e.g., smartphone/desktop):** In a system where a user types in a word processor while listening to music, preemptive scheduling is essential. It ensures *responsiveness* by using clock interrupts to periodically stop running processes. This prevents a single CPU-intensive task from hogging the processor and freezing the interface, ensuring the music doesn't skip and characters appear immediately as typed.

   Real-Time Safety Systems (e.g., Anti-lock Braking System in a car): In safety-critical embedded systems, preemptive scheduling is vital. If a critical sensor detects a wheel locking up, the system must immediately interrupt (preempt) any lower-priority tasks (like updating the dashboard display or radio) to execute the braking control loop.

ii. **High Performance Computing (DTU HPC, weather simulations, scientific calculations):** In a system processing a queue of long calculation jobs without user interaction (like HPC clusters), non-preemptive scheduling is more appropriate. These jobs often consume massive amounts of memory (GBs or TBs), making context switching prohibitively expensive (saving/restoring terabytes of state takes too long and trashes CPU caches). Since *throughput* (jobs per hour) is the goal rather than response time, allowing jobs to run until it finishes or blocks avoids this overhead of frequent context switching. This efficiency allows the CPU to spend more time on the actual computation.

Alternative answer:

ANSWER

i. The OS gives a process some time and if that time passes, it interrupts the process and gives another process a chance to run.

   Web server handling user requests:
   Preemptive scheduling allows the server to handle multiple requests efficiently, ensuring that no single request monopolizes the server's resources. This improves *responsiveness* and ensures that all users get a *fair* share of the server's processing time.

ii. Once a process starts running, it continues until it finishes or voluntarily yields control. There is no fareness, no responsiveness, but there is also no overhead from context switching.

   Embedded systems:
   Embedded systems spend a lot of time idle, waiting for an external event (like a button press). This makes non-preemptive scheduling suitable because the system can run a task to completion without interruption, reducing complexity and overhead.

1.2 Code analysis

Question 22: What will the code in Figure 1 print? Explain briefly your answer. You can assume that all system call invocations are successful. **(4 points)**

ANSWER

```
#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);

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

Figure 1

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. As argued before, child and parent have different physical address spaces, so the child will increment its *own* copy of `counter` and exit, after which it `exit()`s. The parent process waits for any child process to exit (`waitpid(-1,...)`), after which it will print 1, the initial value of `counter`, since the child modified a different copy of `counter`.

Question 23: What will the code in Figure 2 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) `A` variable 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.

Question 24: What will the code in Figure 3 most likely print? Explain briefly your answer. You can assume that all system call invocations are successful. **(4 points)**

ANSWER

```
#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 2

The code will in fact not print anything, since there is a *deadlock*.

Thread 1 acquires the lock on Account 0 and at the same time Thread 2 acquires the lock on Account 1. Thread 1 then tries to acquire the lock on Account 1, but it is blocked since it is already held by Thread 2. Thread 2 then tries to acquire the lock on Account 0, but it is blocked since it is already held by Thread 1. Both threads are now blocked and will never be unblocked, resulting in a deadlock.

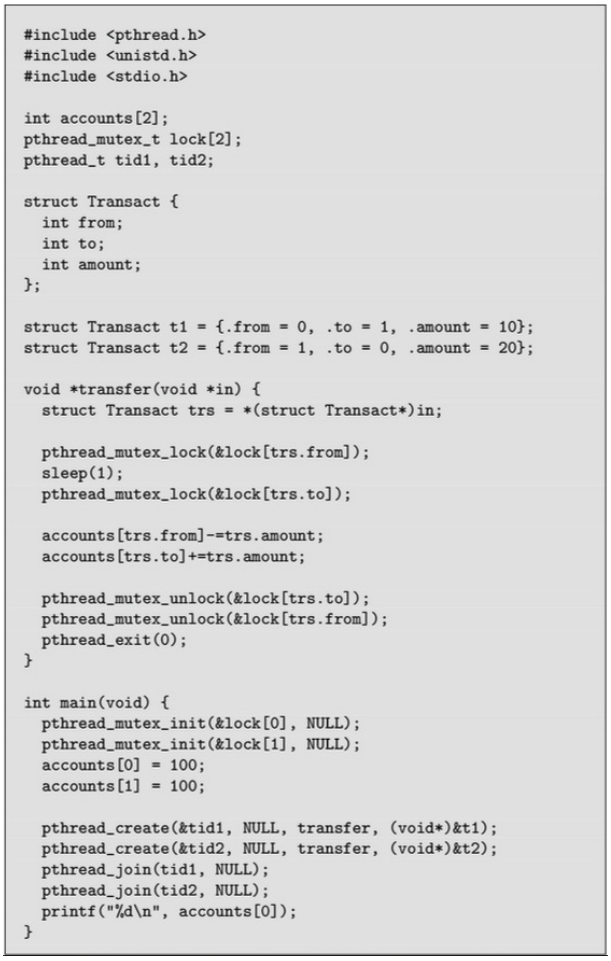
We can fix this by ensuring that the threads always acquire the locks in the same global order, e.g. by checking `trs.from < trs.to`. By forcing all threads to acquire locks in a specific global order (e.g., always lock the lower ID first, then the higher ID), we mathematically guarantee that a cycle cannot exist: If Thread A locks Account 1, Thread B cannot lock Account 2 while waiting for Account 1 to become unlocked, because the rules would have required Thread B to lock Account 1 before trying to lock Account 2.

Question 25: What will the code in Figure 4 print? Explain briefly your answer. You can assume that all system call invocations are successful. **(4 points)**

ANSWER

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. As argued before, child and parent have different physical address spaces, so the child will increment its *own* copy of `counter` and exit, after which it `exit()`s. The parent process waits for any child process to exit (`waitpid(-1,...)`), after which it will print 1, the initial value of `counter`, since the child modified a different copy of `counter`.

Question 26: What will the code in Figure 5 print? Explain briefly your answer. You can assume that all system call invocations are successful. **(4 points)**



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

int accounts[2];
pthread_mutex_t lock[2];
pthread_t tid1, tid2;

struct Transact {
  int from;
  int to;
  int amount;
};

struct Transact t1 = {.from = 0, .to = 1, .amount = 10};
struct Transact t2 = {.from = 1, .to = 0, .amount = 20};

void *transfer(void *in) {
  struct Transact trs = *(struct Transact*)in;

  pthread_mutex_lock(&lock[trs.from]);
  sleep(1);
  pthread_mutex_lock(&lock[trs.to]);

  accounts[trs.from]-=trs.amount;
  accounts[trs.to]+=trs.amount;

  pthread_mutex_unlock(&lock[trs.to]);
  pthread_mutex_unlock(&lock[trs.from]);
  pthread_exit(0);
}

int main(void) {
  pthread_mutex_init(&lock[0], NULL);
  pthread_mutex_init(&lock[1], NULL);
  accounts[0] = 100;
  accounts[1] = 100;

  pthread_create(&tid1, NULL, transfer, (void*)&t1);
  pthread_create(&tid2, NULL, transfer, (void*)&t2);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  printf("%d\n", accounts[0]);
}
```

Figure 3

ANSWER

The code will print `100 000`.

Since the global variable `A` is shared between threads in the same process, all threads will increment the *same* copy of `A`. The mutex ensures that only one thread can access `A` at a time, preventing race conditions.

Question 27: What will the code in Figure 6 print? Explain briefly your answer. You can assume that all system call invocations are successful. **(4 points)**

ANSWER

The code will likely print *nothing* because the program will hang indefinitely due to a *deadlock*.

The `main` function waits for the threads to finish (`pthread_join`) before printing. However, the threads attempt to acquire the two mutexes in reverse order: Thread 1 acquires `lock1` (via `lock_a`) and then attempts to acquire `lock2`. Thread 2 acquires `lock2` (via `lock_a`) and then attempts to acquire `lock1`.

If both threads acquire their first lock before either releases it, they will both be blocked forever

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

int *counter;

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

int main(void)
{
  counter = (int*)malloc(sizeof(int));
  *counter = 1;

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

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

Figure 4

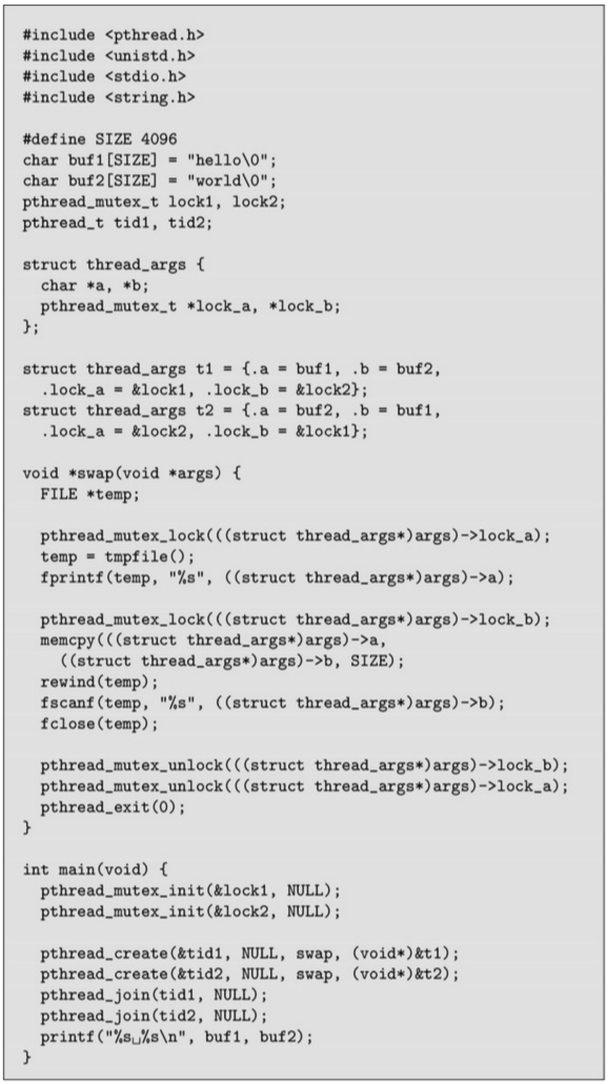
```
#include <stdio.h>
#include <pthread.h>

int A = 0;
pthread_mutex_t m;
pthread_t thread_id[100000];

void* count(void *input) {
  pthread_mutex_lock(&m);
  A++;
  pthread_mutex_unlock(&m);
  pthread_exit(NULL);
}

int main(void) {
  int i;
  pthread_mutex_init(&m, NULL);
  for (i=0;i<100000;i++)
    pthread_create(&thread_id[i], NULL, count, NULL);
  for (i=0;i<100000;i++)
    pthread_join(thread_id[i], NULL);
  printf("%d\n", A);
  return 0;
}
```

Figure 5



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

#define SIZE 4096
char buf1[SIZE] = "hello\0";
char buf2[SIZE] = "world\0";
pthread_mutex_t lock1, lock2;
pthread_t tid1, tid2;

struct thread_args {
  char *a, *b;
  pthread_mutex_t *lock_a, *lock_b;
};

struct thread_args t1 = {.a = buf1, .b = buf2,
  .lock_a = &lock1, .lock_b = &lock2};
struct thread_args t2 = {.a = buf2, .b = buf1,
  .lock_a = &lock2, .lock_b = &lock1};

void *swap(void *args) {
  FILE *temp;

  pthread_mutex_lock(((struct thread_args*)args)->lock_a);
  temp = tmpfile();
  fprintf(temp, "%s", ((struct thread_args*)args)->a);

  pthread_mutex_lock(((struct thread_args*)args)->lock_b);
  memcpy(((struct thread_args*)args)->a,
    ((struct thread_args*)args)->b, SIZE);
  rewind(temp);
  fscanf(temp, "%s", ((struct thread_args*)args)->b);
  fclose(temp);

  pthread_mutex_unlock(((struct thread_args*)args)->lock_b);
  pthread_mutex_unlock(((struct thread_args*)args)->lock_a);
  pthread_exit(0);
}

int main(void) {
  pthread_mutex_init(&lock1, NULL);
  pthread_mutex_init(&lock2, NULL);

  pthread_create(&tid1, NULL, swap, (void*)&t1);
  pthread_create(&tid2, NULL, swap, (void*)&t2);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  printf("%s␣%s\n", buf1, buf2);
}
```

Figure 6

```
#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 7

waiting for the other to release the second lock. Consequently, the threads never exit, `pthread_join` never returns, and the `printf` statement is never reached.

(Note: In the rare event that the operating system scheduler runs one thread entirely to completion before the other thread starts, the output would be "hello world", as the buffers would be swapped twice, returning to their original state).

Question 28: What will the code in Figure 7 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. `pthread_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.

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

ANSWER

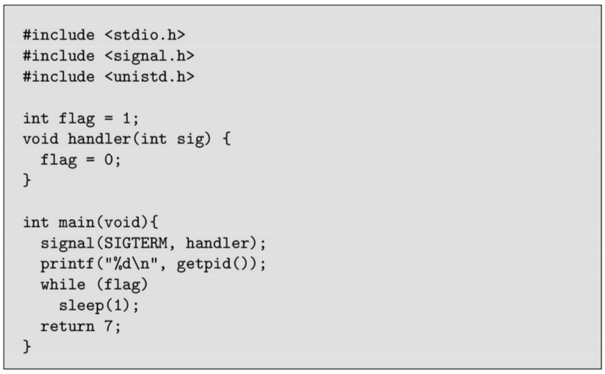
The letter 'a' will be printed 4 times.

The parent 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'.

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

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

Figure 8



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

int flag = 1;
void handler(int sig) {
  flag = 0;
}

int main(void){
  signal(SIGTERM, handler);
  printf("%d\n", getpid());
  while (flag)
    sleep(1);
  return 7;
}
```

Figure 9

Question 30: Assuming the code in Figure 9 is executed and the first print statement prints 20465, which terminal command will make the program terminate with exit code 7? Explain briefly your answer. You can assume that all system call invocations are successful. **(4 points)**

ANSWER

The command `kill 20465` will make the program terminate with exit code 7.

The program prints its PID (20465) and enters an infinite loop. It has registered a signal handler for the `SIGTERM` signal.

We can send a signal to a specific process (based on its process ID) by using the `kill` command. By default, the `kill` command sends the `SIGTERM` signal to the process. Thus, `kill 20465` is equivalent to `kill -15 20465` and `kill -SIGTERM 20465`.

When the process receives this signal, the OS pauses the main loop and executes the `handler` function. This sets the global variable `flag` to 0. Once the handler returns, the `while (flag)` loop condition evaluates to false. The loop terminates, allowing the program to proceed to the final line, `return 7;`, thus exiting with the required code.

Question 31: What will the code in Figure 10 print? Explain briefly your answer. You can assume that all system call invocations are successful. **(4 points)**

ANSWER

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

int main(void){
  int a = 0;
  fork();
  fork();
  fork();
  printf("%d\n", ++a);
  exit(0);
}
```

Figure 10

```
int main(void){
  int a = 0;
  fork();
  printf("%d\n", a);
  fork();
  ++a;
  printf("%d\n", a);
  fork();
  waitpid(-1, NULL, 0);
  exit(0);
}
```

Figure 11

The code will print the number **1** eight times (each on a separate line).

The code calls `fork()` three times. Since every `fork()` creates a new process that is a clone of the caller, the total number of processes becomes $2^3 = 8$.

Because processes have separate, independent address spaces, the variable `a` is not shared between them. Each of the 8 processes has its own copy of `a` initialized to 0. Therefore, every process increments its own local `a` to 1 and prints it.

Question 32: What will the code in Figure 11 print, and what is the exact role of `waitpid` in this example? Explain briey your answer. You can assume that all system call invocations are successful. **(2 points)**

How many distinct copies of the parent process's memory are created during the execution of this code? Explain your reasoning. **(2 points)**

ANSWER

The code will print the number `0` twice and the number `1` four times (total 6 lines of output).

The first `printf` occurs after the first `fork()`, so there are 2 processes. Both print `a` (which is 0). The second `fork()` creates 2 new processes (total 4). All 4 processes execute `a++`, incrementing their own private copy of `a` to 1. Then all 4 execute the second `printf`, printing 1.

Role of waitpid: The `waitpid(-1, NULL, 0)` call instructs the calling process to suspend execution (block) until *any* one of its child processes terminates. This synchronization ensures that parents do not exit before their children, allowing the system to reap the child's exit status and prevent the creation of *zombie processes* (child processes that have terminated but are still in the process table).

Distinct Memory Copies: There are **8** distinct copies of the memory space created.

The code executes `fork()` three times sequentially. Since every `fork()` creates a new process that is a clone of the parent with a *different (physical) address space*, the number of processes grows exponentially ($2^0 \rightarrow 2^1 \rightarrow 2^2 \rightarrow 2^3$). By the end, there are 8 processes running, each possessing its own independent copy of the memory (stack, heap, and data segments).

1.3 You are developing...

Question 33: You are developing an application that is composed of multiple collaborative processes. You wish to implement the following functionality: if a resource is currently unavailable, the process should go to sleep until it receives a wakeup signal from another the process. Which method you would use to avoid a race condition? Explain briefly your answer. **(4 points)**

ANSWER

A semaphore is the perfect fit for this kind of resource management problem. Specifically, a counting semaphore can track resource availability: `sem_wait` automatically puts the process to sleep if the count is zero, and `sem_post` wakes it up if the resource becomes available (count goes from 0 to 1).

Crucially, because these are separate processes (not threads), the semaphore must be allocated in shared memory (e.g. using `mmap`) so all processes access the same synchronization variable.

This is similar to the producer-consumer problem we covered in class. The producer puts items into a buffer, and the consumer takes items out of the buffer. If the buffer is empty, the consumer should go to sleep until the producer adds more.

Question 34: 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, so they can access the same 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.

1.4 Page replacement

Question 35: Which is the key disadvantage of the NFU (Not Frequently Used) page replacement algorithm? 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.

Question 36: Which is the key disadvantage of the FIFO (First-In, First-Out) page replacement algorithm? Explain briefly your answer. **(4 points)**

ANSWER

The oldest page (the one loaded first) might still be useful and heavily accessed.

FIFO removes the page that arrived in memory earliest (like a queue). It does not consider how frequently or recently a page has been accessed. Consequently, it might evict a page that contains critical data or frequently executed code, leading to an immediate page fault.

1.5 Memory management

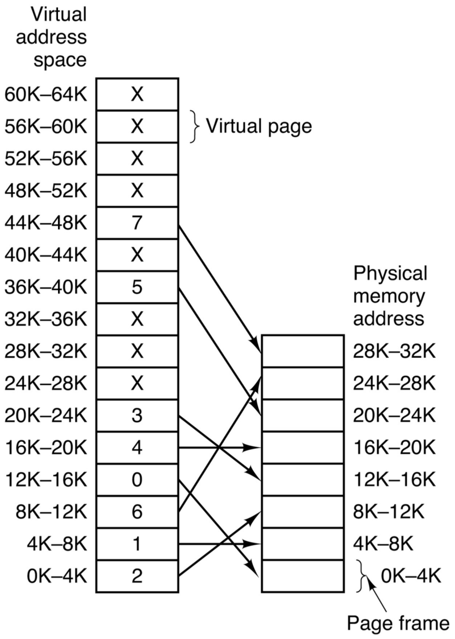


Figure 12

Question 37: Assuming the current state of the memory is as shown in Figure 12, name a virtual address that will generate a page fault if accessed?

Given that the OS uses FIFO as page replacement algorithm and its current list is $[6, 1, 3, 5, 2, 4, 0, 7]$ with the rightmost element representing the tail of the list. What is the physical address corresponding to the virtual address after the appropriate page is loaded? Explain briefly your answer. **(4 points)**

ANSWER

The virtual page from virtual byte address 52 KiB to 56 KiB is not mapped to any physical frame (X in the figure). Hence, any (virtual) byte address in this range, for example byte address 52 KiB + 1 B = 53 249 B, will generate a page fault.

The physical address corresponding to the virtual address 53 249 B is 24 KiB + 1 B = 24 577 B.

FIFO appends to the tail of the list and removes from the head. Because the head is physical frame 6, this has been in memory the longest and will be evicted to make place for the newly loaded page. The newly loaded virtual page will now point to physical frame 6, which corresponds to the physical address range [24 KiB, 28 KiB). We simply add the offset of 1 B to the base address of this physical frame to get the physical address corresponding to the virtual address 53 249 B.

2 Long Questions

Question 1 (12 points)

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

ii. You are developing a web browser. The web browser needs to support multiple parallel tabs, so that the user can browse multiple websites in parallel. Would you implement the browser using multiple processes (one process per tab) or multiple threads (one thread per tab)? Motivate your answer and discuss if the disadvantages of your solution (processes or threads) 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. Web browser implementation:

I would implement the browser using multiple *processes* (one process per tab).

- **Motivation:** The primary requirement for a web browser is stability and security. Web pages often contain buggy scripts or malicious code. If implemented with threads, a crash in one tab would crash the entire browser application. Using processes ensures that if one tab crashes, the others remain unaffected.
- **Relevance of disadvantages:** The disadvantage of higher resource usage (memory overhead per process) is relevant. Processes are "heavyweight" compared to threads.
- **Overcoming them:** Modern operating systems mitigate the creation overhead using ***copy-on-write* (CoW), which allows the child process to share the parent's memory until it modifies it, reducing the overhead of process creation**. To handle necessary communication (e.g., sending rendered data to the main window), efficient IPC mechanisms like *shared memory* can be used.

Question 2 (12 points)

i. Describe, in your own words, how the scheduling algorithm of Linux prioritises the execution of processes that cannot tolerate latency.

ii. Let's assume that you are designing a server for the OS Challenge. Arriving requests have 99% probability to have priority 1 and 1% probability to have priority 200. How would you design parallelism and prioritisation? You shall assume that anything not explicitly specified in this sub-question is as described on the OS Challenge specification document.

ANSWER

i. Linux employs a priority-based scheduling algorithm with 149 distinct priority levels, split between real-time (0-99) processes that and user processes (100-149).

   Processes that *strictly* cannot tolerate latency (e.g., audio processing) are often classified as **real-time**. These processes have absolute priority and are not preempted by standard user processes unless a higher priority real-time process becomes runnable.

   For standard **user** processes that are latency-sensitive (such as interactive applications), Linux employs *dynamic priority levels*. The operating system calculates priority using the formula:

   $$\text{Priority} \propto 1/f$$

   where f is the percentage of the time quantum effectively used by the process.

   Latency-sensitive processes (like text editors) are typically *I/O-bound* and thus spend much of their time waiting for input (blocked state) rather than computing. As a result, they use only a small fraction of their quantum (f is small), resulting in a high dynamic priority. According to the formula $1/f$, this results in a significantly higher dynamic priority, ensuring that when they wake up (e.g., after a keystroke), they are prioritized over CPU-intensive tasks.

ii. As soon as the high priority request arrives, it should be handled immediately. The low priority requests can be queued and processed when there are no high priority requests pending.

   See Long Question 8 for the full implementation.

   Crucially:
   Because high-priority tasks are rare (5%), a Global High-Priority Queue will not suffer from lock contention. Because low-priority tasks are frequent (95%), Local Queues are necessary to prevent the dispatcher from becoming a bottleneck.

Question 3 (12 points)

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

ii. Give one example of a real-world application where you would choose to implement it with multiple threads and one example that you would choose to implement it with multiple processes. Motivate your design decision. Do not use the example applications that we used in class.

ANSWER

i. See Long Question 1.

ii. Multithreaded Application: A High-Performance Photo Editor. *Design:* I would use multiple threads to apply a complex filter (e.g., blur) to a large image. *Motivation:* The threads need to access and manipulate the same large data object (the image pixel array). Since threads share the same address space, they can process different chunks of the image in parallel without the overhead of copying data between memory spaces.

   Multi-process Application: A “grader” for student code submissions. *Design:* A system that receives code uploaded by students, compiles it, and runs tests. I would spawn a new process for every submission. *Motivation:* Student code is untrusted and potentially buggy. It might segfault, enter infinite loops, or try to access forbidden files. By using processes, the system gains isolation. If the student’s code crashes, the grader system remains unaffected. The OS also makes it easier to enforce resource quotas (CPU/RAM) on a specific process to prevent it from hogging the machine.

Question 4 (12 points)

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

ii. You are developing a Domain Name System (DNS) server. The purpose of a DNS server is to translate domain names (e.g. `www.dtu.dk`) to IP addresses (e.g. `192.38.84.35`). 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 (e.g. one process per client) or multiple threads (e.g. one thread per client)? 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. See Long Question 1.

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 server 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 5 (12 points)

i. Summarise the advantages and disadvantages of preemptive and nonpreemptive scheduling.

ii. Elaborate on if, in the OS Challenge, it would be better to process the requests using preemptive or nonpreemptive scheduling?

ANSWER

i. **Preemptive Scheduling:**

- *Advantages:* It prevents a single process from monopolizing the CPU, ensuring fair allocation of resources and better responsiveness for interactive or high-priority tasks. It allows the OS to interrupt a currently running process to switch to a more urgent one.
- *Disadvantages:* It introduces overhead due to frequent context switching and requires hardware support (a timer interrupt). It also complicates resource sharing, necessitating synchronization mechanisms to avoid race conditions.

Non-preemptive Scheduling:

- *Advantages:* It has lower overhead because fewer context switches occur. It simplifies the design of the scheduler and reduces race conditions since processes are not interrupted involuntarily.
- *Disadvantages:* A single process can monopolize the CPU if it does not yield control (e.g., gets stuck in an infinite loop), leading to poor responsiveness and potential starvation for other waiting processes.

ii. **Application to the OS Challenge:**

Preemptive scheduling is significantly better for this challenge.

The scoring mechanism heavily penalizes latency for high-priority tasks: $\texttt{score} = \frac{1}{\texttt{total}} \sum \texttt{delay} \cdot \texttt{priority}$. In a non-preemptive model, if the 4 worker threads are working on low-priority hash calculations (Priority 1), a subsequent high-priority request (Priority 250) would be blocked until all 4 worker threads finish their current calculations. This delay would be multiplied by 250, severely impacting the final score.

However, relying on standard *OS-level* preemption is inefficient because the OS does not know which tasks are urgent. Instead, I implement **voluntary yielding** (user-level preemption):

- Worker threads process low-priority tasks in small chunks (e.g., 1000 hashes).
- Between chunks, the worker performs an *atomic check* on the Global High-Priority Queue size.
- If the queue is non-empty, the worker "yields" by pushing the remainder of its current low-priority task back to its local queue and processing the high-priority task immediately.

This is more efficient than letting the OS handle preemption by creating separate threads for low/high priority tasks and adjusting their priorities with system calls like for example `pthread_setschedparam` (using `SCHED_FIFO`) or `nice()`. While that approach works, it forces the kernel to perform an expensive **context switch** (saving registers, trapping into kernel mode, and polluting the L1/L2 cache) every time a high-priority request arrives. Voluntary yielding achieves the same responsiveness with simple user-space instructions, preserving high throughput.

Question 6 (12 points)

i. Let's assume that you are designing a server for the OS Challenge. Arriving requests have 95% probability to have priority 1 and 5% probability to have priority 250. How would you design parallelism and prioritisation? You shall assume that anything not explicitly specified in this sub-question is as described on the OS Challenge specification document.

ii. Let's assume that you are designing a server for the OS Challenge. You expect to receive an unlimited number of requests, but your memory has room for a cache of only 100 entries. What would be the most efficient cache replacement policy? You shall assume that anything not explicitly specified in this sub-question is as described on the OS Challenge specification document.

ANSWER

i. I would design a **Dispatcher-Worker** model with **Voluntary Yielding** and a **Hybrid Queueing** system, as explained in Long Question 8

Parallelism & Queue Design:
One dispatcher thread receives requests and pushes them to the correct queue based on priority:

- **High Priority (5%):** Since these requests are rare, I would use a single *Global High-Priority Queue* protected by a mutex. The low arrival rate means lock contention will be negligible.
- **Low Priority (95%):** Since these represent the bulk of traffic, a single queue would create a locking bottleneck. I would assign a *Thread-Level* (double-ended) *Local Queue* to each worker thread. This allows workers to pick up 95% of tasks without acquiring a global lock. Furthermore, if a thread runs out of tasks in its local queue, it can steal a task from another thread's local queue, made possible by the double-ended nature of the queues.

Thread Count:

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()`s 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 $\sim 75\%$ 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.

Prioritisation:

The scoring formula is $score \propto \sum \text{latency} \times \text{priority}$. A Priority 250 request is 250 times more detrimental to the score than a Priority 1 request and should be handled as soon as possible. Non-preemptive scheduling is unsafe because a long P1 task could block a P250 task. **Relying on standard OS preemptive scheduling is insufficient because the OS does not know that a queued P250 request is more urgent than a running P1 task**. A long P1 task could block a P250 task, destroying the score. Instead, I implement **Voluntary Yielding**:

- Worker threads split large requests in small chunks (e.g., computing 1000 hashes at a time) *themselves, when needed.* The dispatcher does *not* do this all up front, because then it would hold the lock for a long time.
- Between chunks, the worker performs an atomic check on the Global High-Priority Queue size.
- If the global queue is non-empty, the worker effectively "yields" by pushing its current P1 task back to the head of its local queue and grabbing the P250 task immediately.

ii. The most efficient policy would be the **Clock Algorithm** (Second Chance).

While **LRU** (Least Recently Used) is a better approximation of the optimal algorithm, it requires moving a node to the front of a linked list on **every** cache hit, as well as a hardware timer. In a multi-threaded server, this requires acquiring a write-lock on the cache data structure for every read operation, creating a massive serialization bottleneck even for a small cache of 100 entries.

Why Clock is superior here:

The Clock algorithm approximates LRU with significantly lower overhead. Hash requests (tuples of `(hash, SHA256 pre-image)`) are organized in a circular list with a simple "Referenced" bit (R-bit). On a cache hit, we simply set the $R = 1$. This can often be done *atomically* without locking the entire structure. On a cache miss (eviction), the "hand" sweeps the buffer: if $R = 1$, it sets $R = 0$ (second chance); if $R = 0$, it evicts the entry.

This minimizes locking overhead during standard read operations, maximizing the server's throughput.

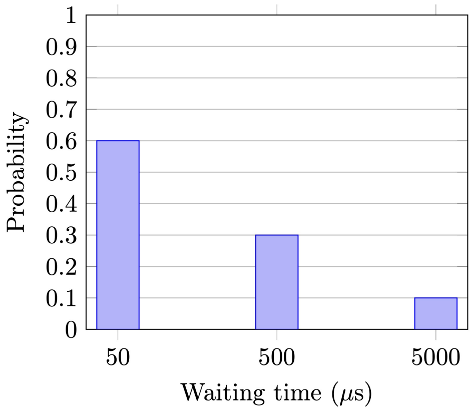


Figure 13: Mutex Wait Times

Question 7 (12 points)

i. Explain in your own words when using a spinlock (i.e., a busy-waiting mutex) in a multiprocessor system can improve the performance.

ii. You are using a multiprocessor system. The operating system implements hybrid mutexes. These mutexes operate as follows. If a thread requests to acquire a locked mutex, the thread first continuously polls the mutex (spins) for a period of time, T. After the time T passes, the thread yields (context switch) and retries when it gets rescheduled. The parameter T is configurable, taking values in μs in the range $[0, 65535]$. Setting $T = 0$ disables spinning. After long-term statistical analysis, you know that the time a thread needs to wait for a locked mutex to be released follows the histogram provided in Figure 13. Moreover, a context switch takes 1000 μs.

Calculate the optimum value for the configuration parameter T, which minimises the overhead (i.e. the sum of time the CPU wastes spinning and switching). For simplicity, you can consider that when the thread gets rescheduled after the first context switch, it finds the mutex unlocked.

ANSWER

i. Spinlock:

Using a spinlock improves performance in a multiprocessor system when the *wait time for the lock is very short*. Specifically, if the time the thread spends spinning is less than the average time required to perform a context switch (saving the current thread's state and loading another), it is more efficient to spin. In this case, waiting for the lock is less expensive than the overhead of blocking the thread, invalidating the cache, and the subsequent delay of rescheduling it.

ii. Optimum value for T:

The goal is to minimize the expected overhead (spinning s + switching c) cost C. We calculate the expected overhead time based on the histogram in Figure 13, assuming two context switches (block and reschedule) take $2c$ time. Below, the first case is "always yield", last case is "always spin".

Intermediate values (e.g., $T = 200$, $T > 5000$) are suboptimal because they increase the spin penalty without catching any new waits.

$$\begin{aligned}
T &= 0 && \implies C = 2c && = 2000\,\mu\text{s} \\
T &= 50 && \implies C = 0.6 \cdot 50 + 0.4 \cdot (T + 2c) && = 850\,\mu\text{s} \\
T &= 500 && \implies C = 0.6 \cdot 50 + 0.3 \cdot 500 + 0.1 \cdot (T + 2c) && = 430\,\mu\text{s} \\
T &= 5000 && \implies C = \mathbb{E}[w] = 0.6 \cdot 50 + 0.3 \cdot 500 + 0.1 \cdot 5000 && = 680\,\mu\text{s}
\end{aligned}$$

So we see that the optimal spinning time is $T^* = 500\,\mu\text{s}$ with a cost of $C^* = 430\,\mu\text{s}$.

Question 8 (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.

Motivation:

The scoring formula (score $= \frac{1}{N} \sum \text{delay} \times \text{priority}$) 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 $\sim 75\%$ 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, the most 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. 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`.
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 (the remainders pushed earlier) 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.

3 New questions

The questions below were not posed in previous years, but were added by me.

Question 1 (12 points)

i. Explain the **Second Chance** page replacement algorithm. How does it improve upon the standard First-In, First-Out (FIFO) algorithm?

ii. While Second Chance is an improvement, implementing it with a standard linear queue can be inefficient. Describe the **Clock** algorithm and explain how it optimizes the implementation of Second Chance.

iii. Consider a system using the **Clock** algorithm with three page frames. The current state is as follows (ordered circularly clockwise):

- **Page A:** Referenced (R=1) (Hand points here)
- **Page B:** Not Referenced (R=0)
- **Page C:** Referenced (R=1)

A page fault occurs. Which page will be evicted? Walk through the steps taken by the algorithm.

ANSWER

i. **Second Chance:** This algorithm addresses the main weakness of FIFO, which might throw out a frequently used page just because it is old. The algorithm inspects the "Referenced" (R) bit of the oldest page (the one at the head of the queue).

- If $R = 0$, the page is old and unused, so it is evicted immediately (just like FIFO).
- If $R = 1$, the page is given a second chance. The R bit is cleared to 0, and the page is moved to the end of the queue (treated as if it just arrived). The algorithm then continues to search for a page with $R = 0$.

ii. **Clock Algorithm:** Implementing Second Chance with a linear list requires constantly moving pages from the head to the tail, which consumes CPU cycles. The Clock algorithm organizes page frames in a **circular list** with a "hand" pointing to the oldest page. When a page fault occurs:

- The algorithm checks the page pointed to by the hand.
- If $R = 1$, it sets $R = 0$ and advances the hand to the next page.
- If $R = 0$, the page is evicted, the new page is inserted in its place, and the hand advances.

This achieves the same logic as Second Chance but avoids the overhead of moving link pointers in the list.

iii. **Execution Steps:**

(a) The hand points to **Page A** ($R = 1$). The algorithm gives it a second chance: sets A's $R = 0$ and advances the hand to Page B.

(b) The hand now points to **Page B** ($R = 0$). This page has not been referenced recently.

(c) **Page B is evicted.** The new page is loaded into Frame B, and the hand advances to Page C.

Question 2 (12 points)

i. Explain the **Least Recently Used (LRU)** algorithm. Why is a perfect hardware implementation of LRU considered expensive or impractical for standard systems?

ii. Describe the **Aging** algorithm. How does it approximate LRU in software, and what specific operations are performed on the counters during each clock tick?

iii. Explain the **Not Recently Used (NRU)** algorithm. Specifically, describe the four classes (0–3) used to categorize pages and how the OS selects a page for eviction.

ANSWER

i. **Least Recently Used (LRU):**
The LRU algorithm assumes that pages used recently will likely be used again soon. It works by evicting the page that has been unused for the longest time.

Why it is expensive:
To implement LRU exactly, the system must track the exact order of access for every single memory reference. This would require maintaining a linked list of pages where a page is moved to the front on *every* memory access, or using 64-bit hardware counters for every page frame. Both approaches impose significant hardware complexity or software overhead (locking and list updates) that slows down the system.

ii. **Aging Algorithm:**
This is an efficient software approximation of LRU. The OS maintains a software counter (e.g., 8 bits) for each page. At every clock tick (periodic interrupt):

- The OS shifts the counter for every page to the right by 1 bit (decreasing its value/weight).
- The R bit (Referenced bit) of the page is added to the Most Significant Bit (MSB) position of the counter.
- The R bit is then reset to 0.

When a page fault occurs, the page with the **lowest counter value** is evicted, as this indicates it was referenced least recently (or least frequently).

iii. **NRU (Not Recently Used):**
NRU categorizes pages into four classes based on their Referenced (R) and Modified (M) bits:

- **Class 0:** Not Referenced, Not Modified ($R = 0, M = 0$). Best to evict.
- **Class 1:** Not Referenced, Modified ($R = 0, M = 1$).
- **Class 2:** Referenced, Not Modified ($R = 1, M = 0$).
- **Class 3:** Referenced, Modified ($R = 1, M = 1$).

The algorithm removes a page at random from the **lowest-numbered non-empty class**. Ideally, it evicts a clean, unused page (Class 0) over a dirty or used page.

Question 3

What is the problem with master-slave multiprocessor systems?

ANSWER

As the number of CPU cores increases, the master CPU becomes a *bottleneck*.

In master-slave multicore systems, one CPU (the master) runs the OS and all user processes run on *slave* CPUs. The master CPU handles all system calls and process management, while the slave CPUs execute user processes. As the number of slave CPUs increases, the master CPU becomes a bottleneck, as it must handle all system calls and process management for all slave CPUs.

This architecture can however work well if all slaves do a lot of local (long) computations.

The alternative to master-slave is *symmetric* multiprocessing, where all CPU cores can execute system calls and processes of the OS. However, in this architecture, there is a need for *mutual exclusion* on OS resources. This can be implemented by either using a *big kernel lock* (lock around the whole OS), or by using *multiple* locks for *independent critical regions* (one mutex for each). The first case is safe, but very *inefficient*, since many parts of the OS are independent (e.g. process table and the file system). The latter case is more efficient, but we have to be careful to avoid *deadlocks*.

Question 4

What is the problem with locking the bus to acquire a lock on a shared variable using TSL in a busy-waiting fashion in a multicore system?

```
do {
    lock_bus();
    lock = TSL();
    unlock_bus();
} while (lock==1);
critical_region();
```

ANSWER

It is safe, but **not efficient**.

In a multicore system, TSL requires the CPU to lock the memory bus to prevent other CPUs from accessing memory during the instruction. Using this in a busy-wait loop causes two major performance issues:

- **Bus Contention:** It constantly locks the bus, which blocks other CPUs from accessing memory and slows them down.
- **Cache Invalidation:** Testing the lock with TSL is always a *write* operation (it writes a 1 to the lock variable). This write invalidates the cache of other CPUs holding that variable, forcing them to reload it from memory and causing further bus traffic.

A better approach is to read the lock variable first without locking the bus, which doesn't invalidate the caches of other CPUs. The CPU attempts the atomic TSL instruction only when the read indicates the lock might be free. Of course, there might occur an interrupt in between, but TSL prevents race conditions after the read operation.

If the variable lock was unlocked and the lock was acquired we can acquire the variable lock and unlock the bus. If the lock was not acquired, we introduce a *delay* between the first TSL and the second one, e.g. using *exponential backoff*.

```
do {
    if (mutex == 1) {
        delay();                    // Exponential backoff
        status = 1;
    } else {
        lock_bus();
        status = TSL(&mutex);    // Returns old value
        unlock_bus();
    }
} while (status == 1);           // Retry if we failed to acquire
critical_region();
```

Question 5

Explain the differences between TCP and UDP.

ANSWER

TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are the two primary transport layer protocols used for communication over IP networks.

TCP is a *connection-oriented* protocol. This means a connection must be established before data can be sent, similar to a phone call. It is reliable: it guarantees that all data will arrive and that it will arrive in the correct order. However, this reliability comes with high overhead due to the mechanisms required to manage the connection (handshake, retransmission, etc.). It is typically used for applications like file transfer, email, and the web.

UDP is a *connectionless* protocol. Communication is based on sending individual messages (datagrams) without a prior connection. It provides a "best-effort" service, meaning there are no guarantees; data may get lost, arrive out of order, or duplicate. Because it lacks these reliability features, it has low overhead and is lightweight. It is typically used for time-sensitive applications like video/audio streaming and DNS queries.

Question 6

Explain the main advantages and disadvantages of *affinity scheduling* and *gang scheduling* in multiprocessor scheduling.

ANSWER

Affinity Scheduling tries to keep a thread running on the same CPU it ran on previously.

- **Advantage:** It is **cache-efficient**. If a thread returns to the same CPU, its data is likely still in that CPU's cache, reducing the need to fetch data from the slower main memory.
- **Disadvantage:** It may conflict with load balancing. Strict affinity could lead to one CPU being overloaded while others are idle. Systems often allow "work stealing" if a CPU goes idle, though.

Gang Scheduling schedules a group of related threads as a single unit to run simultaneously across different CPUs.

- **Advantage:** It optimizes performance for **collaborating threads**. Because all threads in the group start and end their time slices together, they can communicate and synchronize immediately without waiting for peers to be scheduled (reducing "blocking" time).
- **Disadvantage:** It can lead to inefficient CPU utilization (fragmentation). If a "gang" requires fewer CPUs than are available, the remaining CPUs might be left idle during that time slice because they must start/stop together with the gang.

Question 7

Discuss the disadvantages of using memory-mapped I/O registers.

ANSWER

The primary disadvantages of memory-mapped I/O relate to caching and bus architecture complexity:

- **Caching Issues:** Since I/O registers are mapped to memory addresses, the CPU might attempt to cache them like normal RAM. This is disastrous for a status register (e.g., checking if a device is "READY"). The OS/hardware must therefore support selectively **disabling caching** for the specific pages of memory where I/O devices are mapped.
- **Bus Complexity:** In modern systems, the CPU and Main Memory often communicate over a dedicated high-speed bus, while I/O devices reside on a separate, slower bus. If the CPU assumes an address is memory, it puts the request on the memory bus. The I/O device, being on a different bus, will never see this request. To fix this, the hardware requires complex logic, such as the memory controller filtering addresses or the CPU falling back to the I/O bus if the memory bus request fails.

Question 8

Explain a scenario in which **fly-by** mode in a DMA chip cannot be used.

ANSWER

Fly-by mode in a DMA chip cannot be used for **memory-to-memory** and **device-to-device** transfers.

In a standard (two-cycle) transfer, the DMA controller reads data into a temporary internal DMA register in one cycle, and then writes it to the destination in a second cycle. This allows it to address two different memory locations or devices sequentially. Fly-by mode skips the intermediate step: it reads data and writes it directly from the source to the destination. The DMA (Direct Memory Access) chip has an **address** bus and a **DACK** bus (DMA Acknowledge). Fly-by mode places the memory address on the **address** bus and uses the **DACK** (DMA Acknowledge) line to select the I/O device simultaneously. It thereby doubles the effective transfer speed by cutting bus cycles in half (only one cycle per transfer).

However, fly-by mode *cannot* be used for memory-to-memory or device-to-device transfers. There is only *one* address bus and *one* DACK bus. If you want to move data from RAM Address A to RAM Address B, you need the bus to say "Address A" and "Address B" simultaneously. Similarly, when trying to move data from a device to another device, You use DACK to select the first I/O device, but then you would need to use the address bus to select the second device, which is not possible.

Question 9

Explain what *spooling* is and in which kind of I/O software it is used.

ANSWER

Spooling is a technique used to manage access to dedicated I/O devices (such as printers) that cannot be safely shared by multiple processes simultaneously.

If users were allowed to acquire a dedicated device directly, a user might hold onto it indefinitely, blocking everyone else. To solve this, spooling decouples the application from the device. User applications do not talk to the device directly. Instead, they place the data (e.g., a file to be printed) into a special spooling directory. A special background process, called a daemon (e.g., the printer daemon), is the *only* process with permission to acquire and control the physical device. The daemon reads files from the directory and sends them to the device one by one.

Spooling is implemented in *User-Space I/O Software*. Daemons are background services that run in user space, distinct from the kernel-level device drivers or device-independent software.