Technical University of Denmark

02159 Operating Systems
DTU Compute
Fall 2025

DTU

Exam Questions
Operating Systems

Fall 2025

Vincent Van Schependom

Introduction

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.

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 *share variables in memory* and communicate via those variables.

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, even though the virtual address space is the same.

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, which can prevent a process from 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

The DMA is slower than the CPU. If the CPU is idle, it is thus more efficient to let the CPU handle the memory transfer itself at a higher speed.

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).

By offloading the data transfer to the DMA controller, the CPU is free to execute other processes or threads in parallel while the transfer takes place. This improves overall system throughput compared to Programmed I/O.

Additionally, it is efficient in battery-powered embedded systems, as the DMA controller is more energy-efficient (even though it is slower) than the main CPU for moving data.

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. In this case, the watchdog timer will reset the system, such that the OS runs again.

Question 6: 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).

In interrupt-driven I/O, the CPU does not need to continuously check the device's status register. Instead, it can put the calling process to sleep and switch to executing other processes while the I/O device performs its task. The device notifies the CPU via an interrupt only when it is finished, thereby significantly improving CPU utilization and system efficiency compared to *programmed I/O*.

Question 7: 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.

Question 8: 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 9: 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 theseprocesses use the CPU for longer periods.

Question 10: What is the key advantage of monolithic operating systems compared to microkernel operating systems? Explain briefly your answer. **(4 points)**

ANSWER

In monolythic systems, the whole operating system is run in kernel mode, which is faster and more efficient.

In microkernel systems, only the core fundamental services run in kernel mode, while other services run in user mode. They have to use system calls (IPC) to communicate with the kernel, which adds

overhead and can slow down performance.

Question 11: Which POSIX system call should you use to send the `SIGUSR1` signal to a child process? Explain briefly your answer. **(4 points)**

ANSWER

The `kill()` system call can be used to send a signal to a process. It 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)`.

Question 12: 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.

It is responsible for translating virtual addresses to physical addresses. More specifically, it maps *virtual pages* to *physical frames* and enforces logic 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.

Question 13: Name a system operation that is not possible without clock interrupts. Explain briefly your answer. **(4 points)**

ANSWER

Preemptive Scheduling.

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.

Question 14: 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 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: When implementing 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 is held for a very short time (specifically shorter than the time it takes to perform a context switch).

Spinlocks use busy waiting, which wastes CPU cycles. However, putting a thread to sleep (blocking) involves a context switch, which has significant overhead. If the wait time is shorter than the time it takes to perform a context switch, spinning is faster than blocking.

1.2 Code analysis

```
#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

Question 17: 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

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`.

```
#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

Question 18: 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 19: 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)**

```c
#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 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 20: 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

```c
#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

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

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 22: 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 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.

```c
#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

(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 23: 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 24: How many times will the letter 'a' be printed if we execute the code in Figure 4? 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")`

```c
#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

```c
#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

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

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

Figure 8

```c
#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

statement, resulting in 4 prints of the letter 'a'.

Question 25: 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 26: What will the code in Figure 10 print? Explain briefly your answer. You can assume that all system call invocations are successful. **(4 points)**

```c
#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

ANSWER

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.

1.3 You are developing...

Question 27: 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.

Question 28: 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 29: 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 30: 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

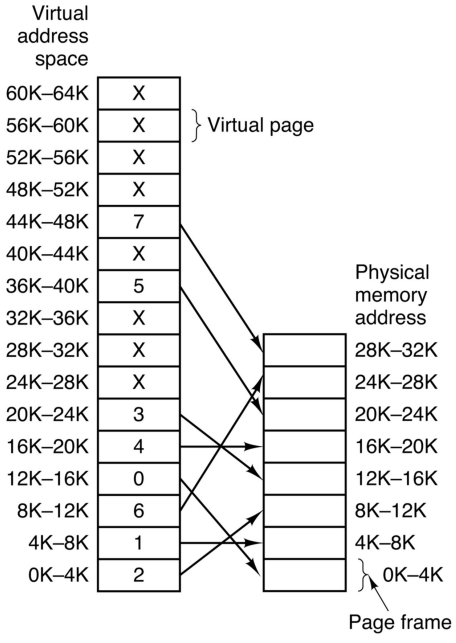

Figure 11

Question 31: Assuming the current state of the memory is as shown in Figure 11, name a virtual address that will generate a page fault if accessed? Explain briefly your answer. **(4 points)**

ANSWER

The virtual page from byte address 24 KiB to 30 KiB will generate a page fault if accessed, because it is not mapped to any physical frame. Hence, a byte address in this range, for example byte address 24 KiB + 1 B = 24 577 B, will generate a page fault.

1.6 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. 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. 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 earlier.

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 3 (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 would likely be better.

The scoring mechanism heavily penalizes latency proportional to this priority level:

$$\texttt{score} = \frac{1}{\texttt{total}} \sum_{i=1}^{\texttt{total}} \texttt{delay}_i \cdot \texttt{priority}_i$$

This means that we should especially aim to keep the latency as low as possible for the highest priority requests.

If a low-priority, high-difficulty request arrives first and occupies a processing thread non-preemptively, a subsequent high-priority request would be blocked until the slow request finishes. This would result in a massive latency penalty for the high-priority request, severely impacting the final score.

A preemptive approach allows the server to interrupt the processing of a low-priority request immediately upon the arrival of a high-priority one, processing the urgent task first. This minimizes the weighted latency and optimizes the score according to the challenge's specific benchmarks.

Question 4 (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 Multithreaded Server with a Thread Pool and User-Level Preemption via Chunking.

Parallelism Design: I would use a thread pool with a number of worker threads equal to the number of available CPU cores (4). This avoids the high overhead of creating/destroying processes (`fork()`) for every request while maximizing CPU utilization.

Prioritization Strategy: Since the score penalizes latency proportional to priority ($score \propto \sum \text{latency} \times \text{priority}$), a Priority 250 request is 250 times more critical than a Priority 1 request. Non-preemptive scheduling fails here because a long-running Priority 1 task could block a Priority 250 task, destroying the score. Conversely, OS-level preemption (processes) introduces too

much overhead: it requires hardware support (a timer interrupt) and complex synchronization mechanisms to avoid race conditions.

Therefore, I would implement **task chunking with voluntary yielding**:

- Maintain two internal queues: a High-Priority Queue (for P=250) and a Low-Priority Queue (for P=1).
- Split brute-force tasks into small "chunks" (e.g., checking 10,000 hashes).
- Worker threads process one chunk at a time. After finishing a chunk of a Low-Priority task, the thread checks the High-Priority Queue.
- If a high-priority request is waiting, the thread yields: it pushes the remainder of its current Low-Priority task back to the Low-Priority Queue and immediately switches to process the High-Priority requests.

ii. The most efficient policy would be **LRU (Least Recently Used)** implemented with a **Hash Map and Doubly Linked List**.

Explanation:

- **Policy:** LRU is widely considered an excellent approximation of the optimal algorithm (i.e., the best possible policy). In a constrained cache of only 100 entries with unlimited incoming requests, ensuring that the most recently accessed (and thus "hot") items are preserved gives the best chance of a hit in realistic scenarios exhibiting temporal locality.
- **Implementation Efficiency:** To make it "most efficient" (fastest), we must avoid $O(N)$ searches. Using a **Hash Map** allows for $O(1)$ lookup to check if a request is cached. Combining this with a **Doubly Linked List** allows for $O(1)$ updates to move accessed items to the front (mark as recently used) and $O(1)$ eviction from the tail when the cache is full. This minimizes the CPU overhead of the caching mechanism itself.

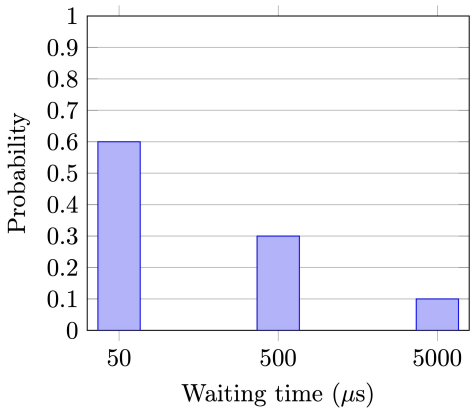

Figure 12: Mutex Wait Times

Question 5 (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 12. 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 12, 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 6 (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:

Work stealing with thread-local double-ended queues.

Motivation:

The current multithreaded server uses a single shared task queue, protected by one *global* mutex. As identified in previous experiments, this centralized queue creates a bottleneck, since all threads fight for the *same global* lock, increasing latency. This was especially noticeable when many "chunks" (ranges within the search space) were assigned at once. I tried fixing this "thread hijacking" by limiting the number of chunks assigned at once (last experiment in the report), but this "batched" approach did not fully solve the contention issue.

Hypothesis:

Implementing *work stealing* with distributed queues will decentralize synchronization and reduce lock contention, improving performance.

Implementation:

Replace the single global queue with a private double-ended queue (`deque`) for each worker thread. Protect each `deque` with its own `mutex` to prevent race conditions when the queue is near empty (e.g., the thread accesses its own queue while another thread is stealing from it).

The main thread accepts an incoming request. It assigns this request to the next worker thread based on round-robin logic, but does NOT split the request (like is done in the current implementation): if the main thread tried to split a range of 1 000 000 hashes into 1000 chunks, it would be stuck in a loop (blocking new connections) and would flood the memory with task structs. Instead, the splitting happens *lazily* (Just-In-Time) by the worker thread.

A worker thread pops a task from the `head` of its own `deque` (LIFO order). It checks if the task range exceeds the defined `CHUNK_SIZE`. If it does, the worker splits the task: it keeps a small chunk to process immediately and pushes the large remainder back onto the `head` of its queue. This keeps the queue depth low while ensuring large tasks are available for stealing.

If a worker runs out of tasks, it scans other threads' queues and acquires their specific `mutex` to steal work from the `tail` (FIFO order), taking the largest available chunks (the remainders pushed earlier) without disturbing the owner's work at the head.

Testing the Hypothesis:

Run a hard `client.sh` script that has a short interval between large requests. If the distributed queue implementation results in higher throughput and lower system-level CPU overhead (measured using `perf` or `top`) compared to our current single-queue baseline, the hypothesis is confirmed.

Alternative answer:

ANSWER

Proposed experiment:

User-Level "Check-and-Yield" preemption.

Motivation:

Our previous experiments presented a contradiction: mathematically, the scoring formula ($score = \sum delay \times priority$) dictates that high-priority requests *must* be processed first to minimize the score. However, Experiment 5 showed that implementing a strict Priority Queue or OS-level Preemption introduced so much overhead (locking, context switching) that performance degraded.

Additionally, Experiment 4 showed that "chunking" (breaking requests into small sub-tasks) was highly efficient. The issue with our current chunking implementation is that once a thread starts working on a low-priority request (split into many chunks), it tends to finish all those chunks before checking the queue again, blocking newer, high-priority requests.

Hypothesis:

We can achieve the score benefits of preemption *without* the overhead of OS context switching by implementing **voluntary preemption**. If a worker thread checks for high-priority tasks in between processing small chunks, it can yield the CPU to urgent tasks immediately, reducing the weighted latency.

Implementation:

We will modify the `Thread Pool + Chunking` implementation:

1. **Dual Queues:** Instead of one complex Max-Heap (which has $O(\log n)$ insertion costs), we use two simple FIFO queues: a `High_Priority_Queue` and a `Low_Priority_Queue`. This keeps queue operations $O(1)$.
2. **Voluntary Yielding:** Currently, a worker thread processes chunks in a loop. We will modify the worker loop so that after finishing a chunk of a *Low Priority* request, the thread atomically checks if the `High_Priority_Queue` is non-empty.
3. **Context Switch:** If a high-priority task is waiting, the thread *yields*: it pushes the remainder of its current low-priority job back to the tail of the `Low_Priority_Queue` and immediately picks up the high-priority task.

Testing the Hypothesis:

We will run the `run-client-highspread-priority-short-delay.sh` script. This script provides the specific workload (high priority spread + short delays) where standard FIFO (Experiment 4) fails to prioritize correctly, but standard (OS-level) preemption fails due to overhead.

If this new implementation yields a lower `score` than the Experiment 5 Priority implementation, the hypothesis is confirmed.

Ultimate answer:

ANSWER

Proposed experiment:

User-level (voluntary) "check-and-yield" preemption with dual (high/low priority) FIFO queues.

Motivation:

The scoring formula (score $= \frac{1}{N}\sum$ delay $\times$ priority) requires processing high-priority requests as fast as possible. My teammates' previous experiments highlighted two critical bottlenecks:

1. **Experiment 5** showed that *OS-level* preemption and priority queues (max-heap) introduce significant overhead ($O(\log N)$ insertions and context switching) that outweighs 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 **user-level preemption** 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 (as opposed to OS-level preemption), we also avoid queue overhead and OS context switches. 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`).

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 **user-level preemption**:

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 the heavy overhead of an OS context switch.

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-level preemption). 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.