Real Time Q&A

Dante Piotto

spring semester 2023

Contents

15 Discuss what kind of analysis you should perform to understand whether a task set is schedulable under dynamic priorities. Consider two cases: relative deadlines equal to periods, or otherwise. **4**

16 The highest locker priority protocol (HLP) is a way of dealing with priority inversion. Discuss its advantages and disadvantages with respect to other resource access protocols seen. **4**

17 Show why EDF is optimal. **4**

18 Discuss the advantages of SRP (Stack Resource Policy) over HLP (Highest Locker Priority). **4**

19 Illustrate a procedure for scheduling asynchronous tasks with precedence constraints. (You do not need to give all the details: just the main idea). Why should such a procedure be preferred to using a generic scheduling algorithm? **4**

20 Can we always determine feasibility of any given set of purely periodic, independent tasks (no shared resources, deadlines equal to periods) under fixed priorities by simply looking at the processor utilization? Why? Explain the concept of least upper bound and when it can help determine feasibility. **4**

21 Define the scheduling problem in general, and identify some special cases where the problem is tractable. **4**

22 Explain the differences between fixed-priority and dynamic-priority scheduling policies. What are the advantages/disadvantages of one over the other? **4**

1 Overview and system structures

1.1 Explain the interrupt mechanism. When does the CPU handle interrupts and what happens when it receives one?

Interrrupts signal events to be handled by the OS, e.g. I/O requests, timer expirations, software errors. As soon as the CPU receives the interrupt, it stops what it's doing, switches to kernel mode and transfers control to an interrupt servis routine (ISR), which is a piece of code designed to handle the interrupt.

1.2 Explain how DMA transfer happens and how it may interfere with program execution

DMA is a mechanism that allows devices to transfer data to and from memory independelty from the CPU, leaving the CPU free to perform other tasks during data trasnfers. However, if the CPU and the DMA device are trying to access the same memory location at the same time, it can lead to conflicts and interfere with program execution.

1.3 Illustrate the main services that an operating system provides to users and processes

Services provided by the OS to the users:

- User Interface
- Program execution: the OS must be able to load programs into memory and run it
- I/O operations: must be controlled by the OS and not directly by the users for the sake of efficiency and protection
- File-system manipulation: read/write, create/delete, search, manage permissions on files

- communications: processes may need to exchange information
- error detection
- resource allocation
- logging: to see which users use what resources
- protection and security

1.4 List the main services provided by a general-purpose OS. Briefly discuss what types of system calls are used to provide such services

types of system calls:
Process control:

- create process, terminate process
- end, abort
- lead, execute
- get process attributes, set process attributes (e.g. pid)
- wit for time, wait for event, signal event
- allocate and free memory

File managment:

- create file, delete file
- open, close file
- read, write, reposition
- get and set file attributes

Device management:

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

Information maintenance:

- get tiome ore date, set time or date
- get system data, set system data
- get and set process, file, or device attributes

Communications:

- create, delete communication connection
- message passing model: send, receive messages to host name or process name
- shared-memory model: create and gain access to memory regions
- transfer status information
- attach and detach remote devices

Protection:

- get and set permissions
- allow and deny user access

1.5 Explain dual mode operation

In order to ensure proper execution of the OS, we need to distinguish the execution of OS code and user code. We therefore use two separate modes of operation (dual mode):

- User mode
- Kernel mode

A bit called the mode bit is added to the computer hardware to indicate the current mode of operation. The system operates in user mode when executing user code, and switches to kernel mode whenever a trap or interrupt occurs, indicating the need for an OS service. Some instructions are designed as privileged, and cannot be executed outside of kernel mode, thus preventing the user from freely executing them in a way that might damage the OS.

1.6 Explain how an operating system is organized, focusing on the concept of dual mode operation and on the services offered by the operating system to processes and users

An OS serves as an intermediary between user and system programs and the hardware. It is made up of user interfaces, system calls and services.

1.7 Explain the architecture of a microkernel-based operating system

The OS is structured removing all non-essential components from the kernel and implementing them as system and user-level programs. The kernel only includes minimal process management, memory management and communication facilities. User modules communicate through message passing.

1.8 What is the main advantage of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?

The main advantages of the microkernel apprach are that the kernel is easier to maintan/extend, as all new services are implemented as user processes in user mode, and it's easier to port to new architectures and more reliable and secure. User programs and system services interact through message passing. The main drawback is that performance can be hindered by the increased system-function overhead communication given by all services being user processes.

1.9 Explain the microkernel operating system architecture. How does the concept of microkernel fit into that of hybrid architectures?

architecture above.

1.10 Illustrate an example of a modern hybrid approach to OS design. Comment on advantages and disadvantages

2 Processes and threads

2.1 Explain what a Process Control Block (PCB) is, and what information it contains. Show some examples of situations where the OS needs to use or modify a PCB

Each process is represented in the operating system by a process control block (PCB), which is a data structure containing various information about the process: Process state, Process Number (PID), Program counter, CPU registers, CPU scheduling information, memory management information, accounting information, I/O status ifnormation. Cases in which the OS needs to use the PCB: process scheduling (PCB contains info such as process priority). Case in which the OS needs to modify PCB: context switch (state is saved into PCB to then be reloaded when the process resumes execution, context switch is performed).

2.2 Explain the main differences between processes and threads

A process is an instance of a program that is being executed, whereas a thread is a flow of execution within a process. A process may contain multiple threads, which allows the process to perform multiple tasks concurrently.

2.3 Illustrate what are the possibilities of communication between processes that execute on the same machine

1. Shared memory: a region of memory to be shared by the communicating processes is established. This requires synchronization
2. pipes: one way communication channel that allows a process to send data to another process. Has a read end and a write end, data written to the write end can be read from the read end
3. signals: message that is sent to a process to notify it of an event.
4. message passing

2.4 Explain the concept of message passing, and what are different realizations of this concept in operating systems

message passing provides 2 operations: send(message) and receive(message). A link between 2 processes is established and messages are exchanged via send and receive. This link can be implemented in a few ways:

2.4.1 direct and indirect communication

In direct communication, processes must name each other: send(P, message) sends a message to process P, receive(Q, message) receives a message from process Q. Links are establised automatically and a link is associated exactly with one pair of communicating processes.
With indirect communication messages are sent and received from mailboxes or ports(e.g. FIFO queues). A link is established when processes share a common mailbox. Two processes may share multiple links.

2.4.2 synchronous and asynchronous communication

By synchronous and asynchronous we mean blocking and non blocking. Blocking send has the sender block until the message is received, blocking receive has the receiver block until the message is available. When both send and receive are blocking we have a rendezvous

2.4.3 buffering

- zero capacity buffer: sender must wait for receiver (rendezvous)
- bounded capacity: sender must wait if the link is full (explicit buffering)
- unbounded capacity: sender never waits (automatic buffering)

2.5 Explain the concept of inter-process communication using message passing, and discuss whether synchronization seems easier or harder when using message passing instead of shared memory

message passing above. Synchronization seems easier when using message passing as opposed to shared memory, because messages can only be read once they are ready, preventing errors caused by modifying shared memory concurrently.

2.6 Give an example of a situation where ordinary pipes are more suitable than named pipes, and an example of a situation in which named pipes are more suitable than ordinary pipes

Example of a situation where ordinary pipes are more suitable than named pipes: when you only need to communicate between a parent and child process: the file descriptors for the read and write ends of the pipe are only accessible to the processes that created the pipe so there is no need to worry about interference from other processes.
Example of a situation where named pipes are more suitable than ordinary pipes: when you need to communicate between unrelated processes, as named pipes can be accessed through their file name or when you want multiple writers to be supported.

2.7 Explain how a client and a server process can communicate by sending messages to each other

A client and a server process can communicate by sending messages to each other through Sockets and Remote Procedure Calls (RPCs)

2.7.1 Sockets

A pair of processes communicating across a network employ a pair of sockets, one for each process. A socket is identified by the concatenation of an IP address and a port. Each port on an IP address(host) corresponds to a mailbox, with a buffer for messages.

2.7.2 RPCs

RPCs are a mechanism that allows a computer program to call a function or procedure that is executed on a differente machine in the network. The client sends a request to the server with the arguments of the function to be executed, the server executes the function with the provided arguments and sends back the return values to the client, that can use them as if it had called a local function.

2.8 Explain the concept of shared memory, and how it applies to processes and threads

Communicating processes establish a region of shared memory, which typically resides in the address space of the process creating the shared memory. Other processes that wish to participate in the communication must attach this region of memory to their address space. As operating systems typically protect the memory of processes from access by other processes, a system call to create an accessible region of memory is required. In order for the shared memory to be accessed safely and consistently synchronization techniques such as mutexes and semaphores are typically used.

2.9 Explain various design choices in message-passing systems, with respect to naming, synchronization, and buffering

Explained above

2.10 Explain the concept of inter-process communication using shared memory, and discuss whether synchronisation seems easier or harder when using message passing instead of shared memory

Done above

2.11 Explain the concept of shared memory, and how sharing memory in Linux differs when we consider processes versus threads

Shared memory done above. Threads belonging to the same process share memory by default, so there is no need to create memory accessible to other processes in such case.

2.12 Explain user threads and kernel threads, and how they are managed by the OS

User threads are supported at the user level and managed without kernel support. Kernel threads are supported and managed directly by the OS. The relationship between user and kernel threads can be implemented in different ways:

2.12.1 Many to one model

Maps many user-level threads to one kernel thread. The entire process blocks if one thread makes a blocking system call, and because only one thread at a time can access the kernel multiple threads are unable to run in parallel on multicore systems.

2.12.2 One to one model

Maps each user thread to a different kernel thread. Provides more concurrency, however creating a kernel thread for each user thread can cause overhead and burden the performance.

2.12.3 Many to many model

Maps multiple user threads to an equal or smaller number of kernel threads. Provides the benefits of one to one, but alleviates its problems.

3 CPU scheduling

3.1 Explain the differences between the three levels of process scheduling

Long term scheduling: selects what processes from a pool of processes that are spooled to mass storage as they can not all be loaded into memory and puts them into memory.
Short term scheduling: selects processes that are ready to execute and allocates the CPU to one of them.
Medium term scheduling: performs swapping: removes processes from memory to either favour a better mix of CPU bound and I/O bound processes, or beacuse a change in memory requirements has overcommitted available memory requiring memory to be freed up.
The main diffrence between these schedulers is frequency of operation: the short term schedulers must operate at very high frequency, and needs therefore to be veryf ast, whereas long term and medium term schedulers are not as time critical.

3.2 Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs? Discuss how existing operating systems can ensure a good response time for interactive processes and a good throughput for CPU-bound processes. You can restrict your answer to on one particular operating system of your choice (among those seen in the course)

The scheduler must distinguish I/O-bound programs from CPU-bound programs in order to ensure a good mix of active processes: if all the processes contending for the CPU are I/O-bound, the ready queue will always be empty and the short time scheduler will have little to do, whereas if they ar all CPU-bound, the I/O waiting queue will always be empty and devices will go unused, making the system unbalanced in both cases.

3.3 Discuss the differences between multitasking and multi-programming

Multiprogramming organizes jobs so that the CPU always has one to execute. Multitasking (or time sharing) is a logical extension of multiprogramming: in time sharing systems, the CPU executes multiple jobs by switching among them so frequently that the user can interact with each program while it is running.

3.4 Draw and illustrate the diagram of process states

Figure 1: Process state diagram

A new process goes into ready state once it is admitted by the long term scheduler, where it waits to be dispacthed by the short term scheduler. Upon dispatch, it goes into running state. Once it is running, one of several events could occur: it could make a request for I/O resources, in which case it goes into waiting state as it waits to be allocated I/O resources, and once it has acquired and used the I/O resources it goes back into ready state. The same thing happens if it waits for an event. If the system is time-sharing, a running process may be preempted or have its time quantum expire, in which case it is put back into ready state. Once a running process terminates, it exits running state and resources are deallocated from it.

3.5 Discuss short-term scheduling, showing the main process states and how multi-programming and time sharing work

Same as above. Multi-programming works by moving a process from the ready state to the running state when the process previously in the running state doesn't need the CPU (because of an I/O request or because it's waiting for an event) to keep the CPU busy at all times. Time sharing works by very frequently moving processes from the running state to the ready state and replacing them with other processes that were ready. This is done by assigning a time slice to each process, which when exceeded causes the process to be placed back into the ready queue.

3.6 Discuss CPU scheduling with the help of a queuing diagram

Figure 2: Queueing diagram

A new process is put in the ready queue. It waits until it is selected for execution (dispatched). Once it is allocated the CPU and is executing, one of several events could occur:

- the process issues an I/O request and is put in the I/O queue
- the process creates a child process and waits for the child's termination
- the process is removed forcibly from the CPU as a result of an interrupt

In all of these cases the process later returns to the ready queue and repeats the cycle until it terminates.

3.7 Among the various schedulers seen that are implemented in mainstream operating systems, choose one that favors interactive processes and briefly explain how that works

The completely fair scheduler, implemented in Linux, is a scheduler that favours interactive processes.
Scheduling is based on scheduling classes. Each class is assigned a specific priority. To select the next process to be run, the scheduler identifies the highest priority process in the highest priority class. Standard linux kernels implement 2 classes: a default class and a real-time class. The scheduler assigns a proportion of CPU time to each task based on the *nice value* assigned to each task, which ranges from -20 to +19, where a lower number indicates higher priority and therefore a higher proportion of CPU processing time. Instead of time slices, the scheduler identifies a targeted latency, which is an interval of time within which every runnable task should run at least once. Priorities are assigned based on the virtual run time of the process. The virtual run time is associated with a decay factor based on the nice value of the task. Virtual run time for a task with nice value ¡0 will be less than the effective run time, while it will be more than the effective run time for tasks with nice value ¿0. The scheduler selects as the next task to be run the task with the lowest virtual run time.

3.8 Present some common aspects of CPU scheduling in general-purpose OSs such as Linux, Windows, and Solaris

3.9 Discuss load balancing in multi-processor scheduling

Load balancing is important to fully utilize the benefits of having multiple processors. It is meant to prevent one or more processors from sitting idle while other processors have high workloads along with processes awaiting the CPU. It is only necessary when each process has its own queue, as in sytems with a common queue, idle processors may take a process from the ready queue and run it. There are two general approaches: push migration and pull migration. With push migration, a specific task periodically checks the load on each processor and if it finds an imbalance "pushes" processes from overloaded processesors to less busy ones. Pull migration instead occurs when an idle processor pulls a waiting task from a busy processor's queue. They are often implemented together.

3.10 Consider a general-purpose operating system. What mechanisms are in place to promote the execution of interactive tasks over batch ones?

4 Synchronization

4.1 Define the critical section problem and show a solution of the problem, which works for two processes and guarantees bounded waiting

Consider a system of n processes $\{p_0, p_1, \ldots, p_n\}$, with each process having a critical section of code (changing common variables, updating tables, writing files, etc.). When one process is in its critical section no other process may be in their critical section. The critical section problem consists of designing a protocol to solve this.

4.1.1 Peterson's Solution

Assumes that the *load* and *store* instructions are atomic. The two processes share two variables: int turn and bool flag[2]. The variable turn indicates whose turn it is to enter the critical section, while the flag array indicates if a process is ready to enter the critical section. For process P_i:

```
FOREVER {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
    CRITICAL section
    flag[i] = FALSE;
    REMAINDER SECTION
};
```

- Mutual exclusion is guaranteed because the turn is either j or i
- Progress is guaranteed as a process only waits if the other on is in the critical section
- Bounded waiting is guaranteed as once a process exits the critical section in may not reenter if the other has already requested access

4.2 Explain what it means to solve the critical section problem, not only in terms of mutual exclusion, but also considering other relevant aspects

discussed in other questions

4.3 Explain the conditions that must be satisfied by a correct solution to critical section problem. For each condition, make a counterexample of a possible (wrong) solution that does not meet the condition

1. Mutual exclusion: if process P_i is in its critical section, no other processes may enter their critical section
2. Progress: if no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter theri critical section next cannot be postponed indefinitely
3. Bounded Waiting: a bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its own critical section and before the request is granted

Counterexamples:

1. using a global variable to control access to the critical section. If the variable is not protected by a synchronization mechanism mutual exclusion is not guaranteed
2.
3. using a spinlock to control access to the critical section. If a process is waiting for the spinlock, all other processes will continuosly spin and try to acquire the lock.

4.4 Explain why a software solution to the critical section problem is in general unsatisfactory

A software solution to the critical section problem is necessarily reliant on certain operations by the operating system being atomic, when they might not be. It therefore cannot guarantee mutual exclusion.

4.5 A possibility for addressing the critical section problem would be to let only one thread execute at a time until the end. Would that solve the critical section problem? Why?

4.6 Motivate the need for synchronization devices provided by operating systems. Then show how semaphores can be used to enable multiple processes to inspect the value of a shared variable, but only one to modify it when no one else is inspecting it. You can use pseudo-code or a known API such as Pthreads or semaphore.h

4.7 Show how semaphores can be implemented to avoid busy waiting

```
typedef struct{
    int value;
    struct process *list;
} semaphore;

wait (semaphore *S){
    S->value--;
    if (S->value < 0){;
        add this process to S->list;
        block();
    }
}

signal (semaphore *S){
    S->value++;
    if(S->value<=0){;
```

```
            remove a process P from S->list;
            wakeup(P);
        }
    }
```

4.8 Show a possible implementation of semaphores using condition variables

4.9 How does the signal() operation on condition variables differ from the signal() operation defined for semaphores?

The signal() operation on condvars can only be called by a process holding the mutex associated with the condvar. Furthermore, signal() for condvars does not change the condition itself, and therefore does not necessarily imply that a process that was waiting for the signal() can continue, whereas the signal() operation for semaphores increments the value of the semaphore, always allowing a process waiting on the semaphore to continue in its execution.

4.10 Briefly explain two different ways of implementing semaphores (with and without busy waiting). For each implementation, show one situation where this implementation is better than the other (i.e., when the implementation with busy waiting is better than the one without, and vice versa)

With busy waiting:

```
    wait(S){
        while (S.value <= 0);
        S.value--;
    }

    signal(S){
        S.value++;
    }
```

without busy waiting:

```
typedef struct{
    int value;
    struct process *list;
} semaphore;

wait (semaphore *S){
    S->value--;
    if (S->value < 0){;
        add this process to S->list;
        block();
    }
}

signal (semaphore *S){
    S->value++;
    if(S->value<=0){;
        remove a process P from S->list;
        wakeup(P);
    }
}
```

can be done with condvars where block() and wakeup() are substituted with pthread_cond_wait() and pthread_cond_signal()

4.11 Show a possible implementation of readers-writers locks using semaphores. Briefly discuss possibilities of starvation

```
int read_count = 0;  // shared variable to count the number of readers
semaphore read_lock;  // semaphore to control access to read_count
semaphore write_lock;  // semaphore to control access to the shared resource

// Function to acquire a read lock
void read_acquire() {
    // Wait for the write lock to be available
    semaphore_wait(write_lock);

    // Increment the read count and release the write lock
    read_count++;
    semaphore_signal(write_lock);
}

// Function to release a read lock
void read_release() {
    // Decrement the read count
    read_count--;
}

// Function to acquire a write lock
void write_acquire() {
    // Wait for the write lock to be available
    semaphore_wait(write_lock);

    // Wait for the read count to be 0
    while (read_count > 0) {
        semaphore_wait(read_lock);
    }
}

// Function to release a write lock
void write_release() {
    // Release the write lock
    semaphore_signal(write_lock);
}
```

4.12 Illustrate a possible solution of the consumer-producer problem, where a producer process and a consumer process have to exchange data via a bounded buffer

4.13 Explain in which situations a lock should or shouldn't be realized as a spinlock, and why

Discussed elsewhere

4.14 Different operating systems provide different synchronization tools. However, there are many commonalities. Briefly discuss what synchronization tools you expect to find in the average operating system, how they function, and what is their purpose

mutex

4.15 Briefly discuss the most commonly available user-level synchronization tools (e.g., conditional variables and various types of locks/semaphores). Briefly describe how they function, and what is their purpose

4.16 Explain why interrupts are not appropriate for implementing synchronization devices in multiprocessor systems. Briefly describe what synchronization devices are provided by an operating system of your choice (among those seen in the course)

Synchronization devices provided in Linux: semaphores, spinlocks in multiprocessor systems that are substituted by disabling and enabling kernel preemption in single processor systems. Also provides readers/writers versions of semaphores and spinlocks.

4.17 Why do some operating systems use spinlocks as a synchronization mechanism only on multiprocessor systems and not on single-processor systems?

Spinlocks do not require a context switch when a process must wait on a lock, however they take up computation cycles that could be used in other ways. On multiprocessor systems one process can spin on a processor while another one executes its critical section on another processor, thus not wasting computing time and saving time that would be used for context switch when control of the lock goes from one process to another.

5 Deadlock

5.1 Why does imposing an order on resource lock acquisition solve the deadlock problem? Discuss

Imposing an order on resource acquisition prevents deadlocks by negating the circular wait necessary condition for deadlocks. Let $R = \{R_1, \ldots, R_m\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to determine an order of resources. Formally, we define a function $F : R \to \mathbb{N}$. We impose the order on resource acquisition by imposing the following: a process can request any number of instances of a resource type R_i. After that the process can request instances of resource type R_j if and only inf $F(R_j) > F(R_i)$. We can prove that this negates the circular wait condition by contradiction: let the set of processes involved in the circular wait be $\{P_0, P_1, \ldots, P_n\}$, where P_i is waiting for a resource of type R_i which is held by P_{i+1}. Then, since P_{i+1} is holding resource R_i while requesting resource R_{i+1}, we must have $F(R_i) < F(R_{i+1})$ for all i. This means that $F(R_0) < F(R_1) < \cdots < F(R_n) < F(R_0)$ and therefore $F(R_0) < F(R_0)$ which is impossible. Therefore, there can be no singular wait.

5.2 Discuss how hold-and-wait-a necessary condition for deadlock-can be prevented, thus effectively preventing deadlock, even if the other conditions may occur

There are two main ways in which the hold-and-wait condition may be prevented:

- we can require each process to request and be allocated all its resources before it begins execution
- we can allow a process to request resources only when it holds none. A process can request some resources and use them, but cannot request any additional resources until the ones it is already allocated have been released.

Both of these protocols have as a drawback that resource utilization may be low, as a process might hold a resource without using it for a long period of time. Another disadvantage is that starvation is possible: a process that needs several popular resources may have to wait indefinitely because at least one of the resources it needs is always allocated to some other process.

5.3 Can a deadlock happen in a system with only 1 thread and 1 resource? Be sure to motivate your answer

A deadlock cannot happen in a system with only 1 thread and 1 resource, as there can be no hold and wait condition, as there are no other processes that can be holding a resource that the process is waiting for.

5.4 Explain a common way of preventing deadlock, which can be applied in cases where some resources must be accessed in mutual exclusion

Imposing an order on resource acquisition, requiring processes to request and be allocated all resources before they begin execution, allowing a process to request resources only when they are holding non, preempting resources from processes.

5.5 Discuss different ways of preventing deadlock, using the dining philosophers as an example (you don't need to show the code)

Making philosophers request and be allocated two chopsticks at a time, having philosophers relinquish a chopstick when their neighbour requests it if they cannot acquire the other chopstick, imposing an order on the acquisition of chopsticks.

5.6 Consider a version of the dining-philosophers problem where the chopsticks are placed at the center of the table, and any two chopsticks can be used by a philosopher. Assume that requests for chopsticks are made one chopstick at a time. What are the necessary conditions for deadlock here? Describe a simple rule for determining whether a particular request can be satisfied without causing a deadlock given the current allocation of chopsticks to philosophers

Deadlock happens if all philosophers hold only one chopstick. A request can be satisfied as long as at least one philospher holds either two chopsticks or none. In this case the situation in which all philosophers hold only one chopstick is avoided.

5.7 Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock free

In order for a deadlock to be possible, there must be a cycle of processes waiting for resources that are held by another process. In this case, since there are four instances of the same resource and three processes that request at most two instances of the resource, at least one process is awarded two instances and can therefore complete and free the resources for the other two.

5.8 Consider two processes and two mutex locks, as in the following piece of code. They don't always lead to deadlock. Describe what role the CPU scheduler plays and how it can contribute to deadlock. Are there ways for the operating system to avoid deadlock?

```
              mutex lock1;
              mutex lock2;

/* process 1 */            /* process 2 */

{                          {
    acquire(lock1);             acquire(lock2);
    acquire(lock2);             acquire(lock1);

    work();                     work();
    release(lock2);           release(lock1);
    release(lock1);           release(lock2);
}                         }
```

The CPU scheduler can cause a deadlock if in scheduling the two processes it has process 1 acquire lock1, then process 2 acquire lock 2. This is a deadlocked state for the system. The Operating System may implement a deadlock prevention avoidance procedure such as Banker's algorithm to avoid deadlock.

5.9 Consider a simple procedure for the allocation of mutually exclusive resources, that immediately satisfies any resource request made by a running process, provided that the resource is available, without considering the other processes. Can you see any potential problems with such a procedure?

Yes, this modality of operation may take the system into an unsafe state and therefore to deadlock.

5.10 Discuss how the possibility of deadlock relates to the presence in the system of multiple instance of resource types

5.11 Consider a system where every resource can be shared by any number of processes at the same time. Do we need to worry about deadlocks? Why?

No, because the mutual exclusivity condition, necessary to deadlocks, is not present.

6 Memory management strategies

6.1 Discuss address binding: how are logical addresses linked to physical addresses?

Binding can happen at 3 stages:

- Compile time: if the location the process must occupy in memory is known at compile time, absolute code can be generated, in which the logical addresses are the same as the physical addresses. If the position of the process in memory changes it will be necessary to recompile the code.
- Load time: If the position of the process in memory is not known at compile time, the compiler must generate relocatable code and final binding must be delayed at least until load time. If the position of the process in memory changes we only need to reload user code.
- Execution time: if the process can be moved during its execution time from one memory segment to another binding must be delayed until run time. This requires special hardware (MMU)

6.2 Why is it important to keep a logical address space distinct from the physical address space? How are the two address spaces used when the process is running?

In order to have a practical way to program, it is beneficial for the programmer to not need to know the physical address of elements of the program. This is important because it allows for efficient mapping of processes into memory, such as dynamic loading, paging, segmentation, swapping. In most operatign systems, the logical address is used to compute the physical address at run time. This system also prevents the user from accessign memory it should not, such as kernel memory or memory of other processes.

6.3 Discuss problems and advantages of contiguous allocation of memory to processes

In contiguous allocation, each process is contained in a single section of memory which is contiguous to the to the section containing the next process. The partitions may be of fixed or variable size. With fixed partitions, the memory is divided into a number of partitions of the same size, and each process occupies one partition. In this case the number of partitions constitutes an upper bound for the degree of multiprogramming. Once a process terminates, it frees its partition and another process may occupy it. This approach is not used nowadays. With variable sized partitions, the OS keeps a table indicating which parts of memory are available and which are occupied. Initially the whole memory is available and is considered a large block called a **hole**. As processes enter the system they are put into an input queue. The OS takes into account the memory requirement of each process and the amount of available memory space and determines which processes are awarded memory. At any given time we have a list of available block sizes and an input queue. Memory is allocated to processes until there are no wholes big enough for the next process. When a process terminates it releases its block of memory, and if it is next to another hole they are merged. When a process is awarded memory from a hole that is larger than its requirement, it takes as much memory as it needs and leaves a smaller sized hole. There are 3 ways to choose a hole for a process: first fit (use the first hole found that is big enough), best fit (use the smallest hole that is big enough) and worst fit (use the largest available hole). First fit and best fit have similar performance in terms of storage utilization, however first fit is generally faster.

Contiguous allocation suffers from external fragmentation. Holes that are too small for any process may form, leading to them being unused. A potential solution is compaction, which consists of reshuffling memory blocks in order to have all the holes be contiguous thus forming one single large hole. Compaction however tends to be quite expensive.

The main advantage of contiguous allocation is its simplicity and the small amount of data the OS need to store for it compared for example to paging.

6.4 Show how segmentation and paging can be combined together, and what are the benefits of their combination. If you wish, you can illustrate using a known memory architecture, such as that of IA-32 systems

The IA-32 architecture supports both segmentation and paging. The CPU generates a logical address, which is sent to a segmentation unit, which in turn generates a linear address, which is sent to a paging unit that computes the final physical address.

6.4.1 Segmentation

Each segment can be up to 4GB, and there can be up to 16K segments per process, divided into two partitions:

- up to 8K segments private to the process (kept in the Local Descriptor Table (LDT))
- up to 8K segments shared among all processes (kept in the Global Descriptor Table (GDT))

Each entry in the LDT and GDT is an 8 byte segment descriptor containing information on a segment, including base and limit. The CPU generates a logical address in the form ¡selector, offest¿, where the selector is a 16 bit element including the segment number, a bit indicating whether the segment is in the GDT or the LDT, and 2 bits used for protection, and the offset is a 32 bit number. The selector is sent to the segmentation unit which uses base and limit to check address validity and produce a 32-bit linear address.

6.4.2 Paging

The linear address is given to a paging unit which generates physical addersses in the main memory. Page sizes can be either 4KB or 4MB. For 4KB pages, a 2 level pagin scheme is used, while a 1 level paging scheme is employed for 4MB pages. The 10 higher order bits of the linear address reference an entry in the outermost page table, which is called the page directory. For 4KB pages, the page directory entry points to an inner page table that is indexed by the next 10 bits in the linear address for 4KB pages. The remaining 12 bits constitute the page offset. For 4MB pages, the 10 highest order bits point directly to the 4MB page frame and the 22 lowest order bits constitute the offset. One entry in the page directory is the `Page_Size` flag, which indicates whether the page is 4MB or 4KB.

6.5 Explain paging and the hardware required to implement it

Physical memory is broken into fixed size blocks called frames and logical memory is broken into blocks of the same size called pages. When a process must be loaded into memory, its pages are loaded into available frames. The backing store (swap partition) is also broken into blocks that are the same size as the frames or clusters of frames. Every address generated by the cpu consists of a page number (p) and a page offset (d). The page number is used as an index into a page table, which contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. Page size is chosen as a power of 2, making the translation of of a logical address into a page number and offset very easy. If the logical address is of size 2^m bytes and the pages are of 2^n bytes, the first $m - n$ bytes of the address designate the page number, while the remaining n bits designate the page offset. Paging is not affected by external fragmentation, however internal fragmentation may happen. The Memory Management Unit looks like in figure **??**

Figure 3: Paging hardware

It must include a page table and registers for the page number, the frame number and the page offset. The page table can also be implemented through a set of registers. All these registers must be very fast to ensure efficient memory access. Using registers is not feasible for large page tables, so the page table is kept in main memory and a page-table base register points to the page table(PTBR). However, in using a PBTR, in order to access a user memory location i, we must first index into the page table, offsetting the value of the PTBR by the page number for i, (memory access) to then access the location in memory given by the page table (another memory access). This requires 2 memory accesses, slowing down memory access by a factor of 2.

The standard solution is to use a translation look-aside buffer (TLB). The TLB is associative high-speed memory. Each entry in the TLB consists of a key (page number) and a value (frame number). When associative memory is presented with an item, it simultanously compares it with all the keys. If it is found, the corresponding value is returned. The TLB can only be quick if it is small, therefore it does not contain the whole page table. However, thanks to the locality principle, a process will likely use a small set of pages. In case of a TLB miss, a memory reference to the page table must be made, and the page and frame number are added to the TLB. Some TLBs store address-space identifiers (ASIDs) in each TLB entry. Each ASID identifies uniquely with a process. When the TLB searches for a key, it also checks the ASID to ensure that the right process is being found, and if the ASID does not match that of the process it is searching for the operation is treated as a TLB miss. If ASIDs are not supported the TLB must be flushed at each context switch.

Figure 4: Paging hardware with TLB

6.6 Explain, with an example, why paging may contribute to the unpredictability of the time it takes to execute a program

Because of page fault rate

6.7 Explain the differences between logical and physical addresses, showing how a mapping can be established between the two when the main memory is managed using paging

discussed elsewhere

6.8 Sometimes strategies that could be very effective in terms of results turn out to be inefficient when implemented, because of the complexity of the data structures and mechanisms involved. Such is the case for some page replacement methods. Illustrate this point using a page replacement method of your choice

This statement applies to the Least Recently Used algorithm for page replacement. There are two feasible implementations:

- Counters: we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented at every memory reference, and whenever a reference to a page is made, the value of the clock is copied into the time-of-use field. We replace the page with the smallest value. This scheme requires a search of the table and a write to memory for each memory access.
- Stack: a stack of page numbers is used, and whenever a ppage is referenced it is removed from the stack and put on top. This way the most recently used page is always on top and the least used is on the bottom. Updating the stack is more costly, but there is no search for a replacement.

In any case, special hardware is required to make both implementations viable, as the absence of it would signify an unacceptable slow down of the memory system.

6.9 Explain what happens when there are not enough frames assigned to an active process, and what can be done to face this problem

When there are not enough frames assigned to an active process, it incurs in a high page-fault rate, leading to thrashing, where the process spends more time paging than executing. In this case the OS should allocate more frames to the process, for example by reducing the degree of multiprogramming through the medium term scheduler.

6.10 What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate the problem?

The cause of thrashing is not enough frames being allocated to a process. The system detects thrashing by monitoring the number of page faults: when it is high, thrashing is occurring. Once the OS detects thrashing, it should award the process more frames by reducing the degree of multiprogramming through the medium term scheduler.

6.11 Explain paging on demand. What is the consequence of a page miss?

Demand paging works by brigning a page into memory only when it is needed. It can be implemented with a page table including valid/invalid bits. The bit is invalid if the page is not loaded into memory. When an invalid page is references a trap is generated that triggers the loading of the page into main memory (page fault). On page fault:

- an internal table is checked to see if the reference was valid
- if it was invalid, the process is terminated. If it was valid but the page was not in memory, it is paged in
- a free frame is found
- a disk operation is scheduled to read the desired page into the newly allocated frame
- when the disk read is complete, the internal table is is modified to indicate that the page is now in memory
- the instruction that was interrupted by the trap is restared

Hardware needed to support demand paging: page table with valid/invalid bit and secondary memory to hold pages outside main memory (swap space).

6.12 Discuss what happens with virtual memory when a process needs to access a page that is not in the memory

question above

6.13 Show how page fault rate affects the performance of paging

With demand paging:
with an average page-fault service time of 8 milliseconds and a memory access time of 200 nanoseconds:

$$\begin{aligned}\text{effective access time} &= (1-p)\times(200) + p(8\text{milliseconds})\\ &= (1-p)\times 200 + p \times 8,000,000\\ &= 200 + 7,999,800 \times p\end{aligned}$$

Effective access time is directly proportional to page-fault rate. If we want performance degradation to be less than 10 percent:

$$\begin{aligned}220 &> 200 + 7,999,800 \times p\\ 20 &> 7,999,800 \times p\\ p &< 0.0000025\end{aligned}$$

That is, only one access out of 399,990 can page-fault. Therefore page-fault rate must be kept extremely low with demand paging

6.14 Discuss frame allocation: how can we determine how many frames should be allocated to an active process?

There are various methods to allocate frames:

- Equal allocation: the available frames (after OS frames have been allocated) are split equally between the processes
- Proportional allocation: frames are allocated to processes proportionally to their size. Let s_i be the size of process p_i, and let $S = \sum s_i$. Then, if the total number of frames is m, we allocate a_i frames where $a_i = s_i/S \times m$. Naturally, a_i needs to be adjusted to be at least as big as the minimum number of frames required by the instruction set.
- Priority allocation: a proportional allocation scheme using priorities rather than size

We can also consider how we replace pages. There are two categories:

- global replacement: a process selects a replacement frame from the set of all frames. Allows a high priority process to icrease its own frames at the expense of a lower priority process. Inconsistent per process performance because performance depends on other processes, but overall higher throughput.
- local replacement: a process can only select a frame from its own set of allocated frames. The number of frames is fixed for all processes. Consistent per process performance, but processes might take up frames without using them much leading to lower throughput.

6.15 Explain how the concept of locality helps allocating each process the right amount of physical memory

The assumption of locality is exploited by the working set model. The model uses a parameter, Δ to define a working set window. The working set is the set of pages in the most recent Δ page references. Once a Δ has been selected, the OS monitors the working set of each process and allocates to that proces enough frames to provide it with its working set. If there are extra free frames, another process can be initialized. If the sum of the workin set frames exceeds the number of available frames, the OS can suspend a process. This prevents thrashing while ensuring the maximum degree of multiprogramming, thus optimizing CPU utilization.

6.16 Show how virtual memory works

6.17 Explain copy on write and why such a technique can improve performance

With copy on write, a child process shares the same pages as its parent on creation. If either process modifies a shared page, only then the page is copied. This helps performance because it delays the creation of physical space for a new process, which would require both time and resources(physical memory)

6.18 Discuss one way to allocate kernel memory

The buddy system allocates memory from a fixed size segment consisting of physically contiguous pages. Memory is allocated using a power of 2 allocator: it satisfies requests in units sized as powers of 2 (each request is rounded up to the nearest power of two). If a segment of memory is too large, it is divided into two equal sized buddies. This is repeated until we have two buddies that are of the appropriate size, then one is allocated. An advantage of the buddy system is that that adjacent buddies can be combined to form larger segments very rapidly using a techinque known as coalescing. The obvious drawback is that this is subject to internal fragmentation

Figure 5: Buddy system allocation

6.19 Consider allocation of kernel memory. Why is paging not a good idea? How can memory be allocated to the kernel, without paging?

Paging is not a good option because some kernel data structures need to be allocated contiguosly. Furthermore, the kernel requests memory for data structures of varying sizes, some of which are less than a page in size. Thus, to avoid internal fragmentation, paging is to be avoided. Buddy allocation and slab allocation explained elsewhere

6.20 Explain slab allocation of kernel memory. Why is paging not really used for kernel memory allocation?

A slab is made up of one or more physically contiguous pages. A cache is made up of one or more slabs. There is a single cache for each unique kernel data structure (e.g. semaphores, PCBs, etc.). Each cache is populated with objects that are instantiations of the kernel data structure the cache represents. When a cache is created, a number of objects (which are initially marked as free) are allocated to it. Initially all objects in the cache are marked as free. When the kernel needs a new object for a data structure, the allocator can assign any free object from the cache to satisfy the request, and the object is marked as used. A slab may be full (all objects are used), empty (all objects are free), or partial. The slab allocator tries to satisfy requests with a free object from a partial slab, then from an empty slab. If no empty slab exists, a new one is allocated from contiguous Physical pages and assigned to a cache.

Benefits: no internal fragmentation, and slots of the needed size are created in advance so memory requests can be quickly allocated.

7 I/O subsystems

7.1 Linux makes a design choice to manage files and I/O devices in similar ways at the system call level. Briefly explain what that means, and discuss the advantages of such a design choice

7.2 Explain the concept of device drivers and how they relate to the rest of the system

A device driver is a piece of software of the OS the purpose of which is to handle I/O operations. To start an I/O operation, the device driver loads the appropriate registers within the device controller. The device controller reads the registers, performs the I/O operation, and when it is done informs the device driver via an interrupt. The device driver then returns control to the OS. The device driver sits inbetween the OS and the device controller, ensuring proper communication between them, presenting a uniform device access interface to the I/O subsystem.

7.3 Explain how the operating system can manage data exchange with external devices

7.4 Explain the differences between blocking, non-blocking, and asynchronous I/O

When a blocking I/O system call is made by an application, it is suspended by the OS and moved into the wait queue. After the system call completes the application is moved back to the run queue, and when it resumes execution it will receive the values returned by the system call.

With a non-blocking I/O system call, the execution of the application is not halted for a long time, and instead it returns quickly with a return value that indicates how many bytes were trasnferred.

An asynchronous I/O system call returns immediately, without waiting for the I/O to complete. The application continues to execute its code, and whenever the I/O transfer is complete it is communicated to the application, either through the setting of some variable in the applications address space, a software interrupt or through a call-back routine that is executed outside the linear control flow of the application.

7.5 Explain how a device driver and a device controller interact

Explained elsewhere

7.6 Explain how the interaction between controller and host happens, by briefly describing the notions of I/O port and its registers, and the main differences between busy-waiting and interrupt-driven I/O

7.7 Explain how devices are managed by an operating system such as Linux