Process alone

1. Explain what is a process.

Pokud o procesech mluvíme samostatně (nemáme vlákna), myslíme tím kód a stav (paměť, registry, datové struktury OS, prostředky, …). V případě, že máme v systému jak procesy, tak vlákna, je proces pouze pasivní obálka na vlákna (Zůstává mu pouze správa „paměti“ a možná handle na některé prostředky OS).

2. Explain how the operating system or the first application process gets started when a computer is switched on.

V HW je napevno zadrátována adresa, od které se začínají vykonávat instrukce procesoru. Na této adrese se nachází tzv. bootstrap program, který je vypálený v některé z pamětí (eprom, rom, apod.), jehož úkolem je z nějaké pevně dané adresy načíst data do paměti.

3. Explain what it means to relocate a program and when and how a program is relocated.

Nastává během nahrávání programu do paměti a znamená posunutí programu (instrukce a statická data) v paměti. Pokud je program napsaný s absolutními adresami, musel by běžet pouze v určité části paměti (a to nechceme; u relativní adresace problém není, tam posouváme pouze base). Samotná relokace spočívá v tom, že se z relokační tabulky (kterou vytváří linker) berou jednotlivé odkazy do kódu na adresy, které je třeba změnit a k nim se připočte base register.
Program exe = Kód + statická data + relokační tabulka.

4. Explain what information is needed to relocate a program and where it is kept.

Relokační tabulka je tabulka s relativními pointery na adresy v kódu, které je nutné opravit při relokaci adresy. Například v EXE se nachází před kódem na konci hlavičky; Dále se může vyskytovat v ELF hlavičce v sekci rel.

5. Explain what it means to link a program and when and how a program is linked.

Pomocí programu, který se jmenuje linker. Statické (vše na jednom místě) a dynamické (portabilita) linkování. Na začátku: image, které obsahují EXPORTS (nabídka), a EXTERNALS (požadavky); tyto image se slijí do jednoho.
– resolve externals znamená průchod tabulkami a nahrazování referencí z jednoho souboru do druhého.

6. Explain what information is needed to link a program and where it is kept.

EXPORTS a EXTERNALS. Ve formátu binárky ELF jsou to např. sekce symtab a dynsym pro statické a dynamické linkování (sekce obsahující tabulku symbolů).

7. Explain what the interface between processes and the operating system looks like, both for the case when the operating system code resides in the user space and for the case when the operating system code resides in the kernel space.

Volání systému jsou zabalena do knihovních funkcí; používají se tedy jako klasické funkce. Syscall je speciální rutina s vyššími právy, volaná většinou kódem s nižšími právy, obsahuje kontrolu, zda vůbec dovolí rutinu spustit (kontrola ukazatelů, práv, a další). Speciální instrukce procesoru SYSENTER a SYSEXIT, nebo starší Int 2e (před těmito instrukcemi předáme parametry do těch správných registrů). Rozhraní musí být **efektivní** (rychlé přepínání módů), **robustní** (kvůli škodlivému kódu) a **flexibilní** (aby se později při přidání nových služeb nemuselo měnit).

8. Propose an interface through which a process can start another process and wait for termination of another process.

```
int process_create(process_t *proc_ptr, void * img, const size_t size)
int process_join(process_t proc)
int thread_create(thread_t *thr_ptr, void*(*main)(void*), void* data, const
uint32_t flags)
int thread_join(thread_t thr)
```

Achieving parallelism

9. Explain how multiple processes can run concurrently on a single processor hardware.

Princip spočívá v tom, že se různé procesy rychle střídají na jednom procesoru a tím vzniká iluze paralelismu. Během výměny procesů je nutné uložit kontext aktuálního procesu tj. jeho stav – registry (SP, PC, všeobecné registry), případně věci okolo paměti (záleží na architektuře, pokud existují pouze procesy, tak i paměť, …) na nějaké „bezpečné místo“, například na zásobník procesu a obnovit kontext procesu, který střídá aktuální. Ten následně pokračuje ve vykonávání svého kódu.

10. Explain what is a thread and what is the relationship between threads and processes.

Vlákno je běžící program, aktivita uvnitř procesu, akt vykonávání programu, naproti tomu je proces pasivní obálka, poskytující běžícím vláknu uvnitř zdroje, kód a paměť. Vláken v jednom procesu může být více, sdílejí paměť a kód, ale mají své vlastní zásobníky.

11. Explain what happens when the thread context is switched.

Uloží se aktuální obsah registrů (všeobecné registry, flags, SP, PC) a poté se načte uložený obsah registrů nového vlákna. V principu jde tedy pouze o přepnutí na jinou sadu registrů.

12. Explain what happens when the process context is switched.

- Během volání OS je uložen PC na zásobník, procesor ukládá originální hodnotu na zásobník toho procesu, který bude odstaven.
- OS se ujme řízení, uloží ostatní registry na zásobník odstavovaného procesu.
- OS přepne zásobník na zásobník procesu, který bude pokračovat v běhu.
- Ze zásobníku se obnoví registry.
- Jako poslední se obnoví PC, tedy ukazatel na instrukci, kterou se bude pokračovat v běhu programu, na jehož kontext OS právě přepnul.
- Navíc je požadavek znepřístupnit paměť aktuálního procesu jiným procesům (řeší se stránkováním a přepnutím do jiného adresáře stránek)
- Další kontext je kontext ostatních zařízení, ale protože se k nim přistupuje většinou nepřímo, tak se neukládá přímo (a navíc jsou uzamčeny), tak to není třeba. Pouze u přímého přístupu k zařízením bych si musel uložit i jejich kontext.

13. Na popisu přepnutí kontextu krok po kroku ukažte, jak se implementace vláken v uživatelském prostoru a implementace vláken v prostoru jádra liší ve způsobu přepínání kontextu.

Pro vlákna je důležité zajistit jim přepínání kontextu, zásobník a přístup k procesoru. Je tedy možné naprogramovat vlákna v user-space.
user threads jsou naprogramovány aplikací:
+: šetří se režie na volání kernelu, šetří se paměť kernelu, lze naimplementovat „na míru“, přepínání vláken může být kooperativní (protože kernel o těchto vláknech neví).
– pokud některé vlákno zavolá kernel a zůstane v něm, nastává lock; vlákna nemůžou bez podpory kernelu využívat více procesorů

kernel threads jsou vlákna uvnitř kernelu, nebo vlákna spravované kernelem.
+: lze uzpůsobit plánování vláken na konkrétní procesory, není problém s odnímáním běhu jednotlivým vláknům, není problém s reentrantností knihoven (malloc apod.).
–: je to napevno navržené, nelze doimplementovat další features.

14. Explain how the requirements of interactive and batch processes on the process scheduling can contradict each other.

An interactive application spends most of its time waiting for input. When the input does arrive, the application must react quickly. It is often stressed that a user of an interactive application will consider the application slow when a reaction to an input does not happen within about 150 ms.
A batch application spends most of its time executing internal computations in order to deliver an external output. In order to benefit from various forms of caching that happen within the hardware and the operating system, a batch application must execute uninterrupted for some time.

15. List the typical requirements of an interactive process on the process scheduling.

- *responsiveness* – reakce na to, že se něco děje, reakce na asynchronní události – user input, network requests apod.
- *fairness* – spravedlivost, procesy s podobnými parametry (priorita, časové kvantum) budou také podobně plánovány a žádný nebude upřednostněn.

16. List the typical requirements of a batch process on the process scheduling.

- *efficiency* – efektivní využití a rozložení zdrojů (například plánování procesů na více procesorů).

17. List the typical requirements of a realtime process on the process scheduling.

- *predictability* – předvídatelnost, systém garantuje určité chování scheduleru, je důležité pro realtime aplikace.
- *turnaroud* – co nejmenší délka úlohy, aby nikdo neblokoval ostatní.

18. Explain the difference between soft and hard realtime scheduling requirements.

- hard realtime – úloha se MUSÍ stihnout do daného limitu.
- soft realtime – úloha by se měla stihnout do daného limitu, ale je povolen statistický limit prošvihnutí úloh (třeba jedna úloha ze sta).

19. Define typical phases of a process lifecycle and draw a transition diagram explaining when and how a process passes from one phase to another.

- 1) Init; 2)Ready; 3)Running in kernel; 4) Running in user; 5) Waiting; 6) Zombie
- 1)->2); 2)=>3) schedule; 3)=>2)Yield, Preemption; 3)=>5) disk, mutex; 5=>2); 3)=>4) return; 4)=>3) Yield [Syscall/Exception], interrupt; 2), 3)=>6) Kill
- [stejné, jako u nás; v prezentaci máme ale jen jeden running stav, ve skutečnosti jsou dva].

20. Explain cooperative context switching and its advantages.

Procesy se střídají na procesoru dobrovolně, bez řízení OS.
+: menší režie, nejsou nevhodná přeplánování (pokud k nim vůbec dojde).

21. Explain preemptive context switching and its advantages.

Procesy jsou plánovány OS. Ten určuje nejen kdo „poběží“, ale i horní odhad času obrátky.
+: Můžu nastavit pravidla přidělování času procesoru, robusntější systém odolný proti záměrně „náročným“ programům.

22. Explain the round robin scheduling algorithm by outlining the code of a function GetProcessToRun that will return a process to be scheduled and a time after which another process should be scheduled.

Kruhová fronta připravených procesů, plánovány jsou ty ze začátku fronty, po ukončení nebo vypršení časového kvanta jsou řazeny na konec (a kód je jednoduchý).

23. Explain the simple priority scheduling algorithm with static priorities by outlining the code of a function GetProcessToRun that will return a process to be scheduled and a time after which another process should be scheduled.

Více RR front, na každou prioritu jedna. Když je proces přeplánován před vyčerpáním časového kvanta, je na začátek fronty, jinak na konec. Vždy se vybere první vlákno z první neprázdné fronty té nejvyšší priority.

24. Explain the simple priority scheduling algorithm with dynamically adjusted priorities by outlining the code of a function GetProcessToRun that will return a process to be scheduled and a time after which another process should be scheduled.

Více front RR, pokud je proces přeplánován před vypršením svého kvanta, vloží se na konec fronty priority o jedna vyšší, jinak do jeho priority. Vždy se vybere první vlákno z první neprázdné fronty té nejvyšší priority.

25. Explain the earliest deadline first scheduling algorithm by outlining the code of a function GetProcessToRun that will return a process to be scheduled and a time after which another process should be scheduled.

Tupě prioritní RR, proces s nejnižší deadline vždy musí doběhnout jako první. Řazení tedy podle deadline.

26. Explain the function of the Solaris process scheduler by describing how the algorithm decides what process to run and for how long.

- priority 0-159, pro každou RR.
- Třídy procesu:

timesharing — 0-59 (nižší priority mají delší kvantum), priorita se zvyšuje při čekání na něco nebo když proces nespotřebuje své kvantum, priorita se snižuje po spotřebování kvanta, změny priority určuje tabulka. Podle priority se také určí délka kvanta (čím větší, tím méně a naopak).

system — 60-99, Statické priority. Procesy s kritickými prostředky.

realtime — Používá priority 100-159, priorita procesu a přidělované kvantum se nastavuje syscallem priocntl a od okamžiku nastavení se nemění.

Vybírá se proces s nejvyšší prioritou, preemptnutí kernelu jen v několika dobře definovaných okamžicích (abych nezpůsobil nekonzistenci). Poskytuje volání, které požádá kernel o dočasné neodebírání procesoru a umožňuje vytvářet vlastní Schedule.

27. Explain the function of the Linux process scheduler by describing how the algorithm decides what process to run and for how long.

Linux 2.4:
Dvě třídy procesů: normal a realtime
Realtime plánování FIFO/RR, *goodness value* mají zvýšenou o 1000
Aktuální priorita je počet tiků, které procesu zůstávají.
Po vyčerpání kvanta se jejich priorita přepočítá podle nice value.
Linux <2.6.23:
časová složitost plánovače je O(1).
priority 1-140 , pro každou dvě RR fronty (active a expired).
program je v active frontě, pokud byl odstaven před vypršením svého kvanta, po vypršení kvanta je zařazen do expired fronty. Vybírá se fronta s nejnižší prioritou, active fronta má přednost před expire je-li active fronta prázdná a v expired něco je, fronty se prohodí.
Podpora metriky pro interaktivitu – počítá se podle toho, jak dlouho proces čeká na IO; ty co čekají na IO jsou odměněny, ty které hodně počítají, jsou penalizovány.
program co čeká na IO se z active fronty nevyhazuje.
Linux >=2.6.23:
Zohledňuje se afinita. Podpora realtime. Pro aplikace Completely Fair Scheduler – (RB strom procesů; setříděný dle sežraného času procesoru).

28. Explain the function of the Windows process scheduler by describing how the algorithm decides what process to run and for how long.

- třídy priorit IDLE, NORMAL, HIGH, REALTIME
- Uvnitř každé třídy relativní priorita IDLE, LOWEST,BELOW_NORMAL, NORMAL, ABOVE_NORMAL, HIGHEST, TIME_CRITICAL
- třída priorit (pro proces), relativní priorita (pro vlákno).
- boost po: dokončení IO, přepnutí na foreground, probuzení, foreground vlákna.
- 0; 1-15 (zde je možný boost); 16-31 (realtime, zde není možný boost).
- plánuje se thread s nejvyšší prioritou, na stejné prioritě funguje round robin
- zohledňuje se afinita na SMP
- plánují se thready, nikoli procesy
- datové struktury: round robin pro každou prioritu, 32-bitová maska označující neprázdné fronty, 32-bitová maska volných CPU

29. Define processor affinity and explain how a scheduler observes it.

Svázanost procesu k procesoru. Pokud proces běžel na nějakém procesoru, tak v jeho registrech, cache apod. zůstala pravděpodobně užitečná data. Proto je vhodné při dalším plánování toho procesu jej umístit na naposledy navštívený CPU. Toto si může pamatovat například plánovač. Ve windows je součástí struktury vlákna (ideální procesor (cyklicky generován) a poslední procesor), tyto procesory se zohledňují při výběru procesoru, na kterém bude vlákno běžet příště.

30. Propose an interface through which a thread can start another thread and wait for termination of another thread, including passing the initial arguments and the termination result of the thread.

```
int thread_create (thread_t * thread_ptr, void (* thread_start) (void *),
void * data)
int thread_join (thread_t thr)
int thread_join_timeout (thread_t thr, const unsigned int usec)
```

Process Communication

31. Propose an interface through which a process can set up a shared memory block with another process.

```
void *mmap (void *start, size_t length, int prot, int flags, int fd, off_t
offset);
adresa kam se to má namapovat, velikost, ochrany (R/W), flags (SHARED,
PRIVATE – Copy on write), file descriptor souboru a offset v souboru.
int munmap (void *start, size_t length);
```

32. Define synchronous and asynchronous message passing.

- *Synchronní* - Když se vrátím z funkce, co zprávu zaslala, vím, že byla zpráva také přijata.
- *Asynchronní* – Vím akorát, že byla zpráva odeslána.

33. Define blocking and non blocking message sending and reception.

- *blokující* – procedura co zprávu odeslala nebo obdržela, čeká na vyprázdnění odesílacího bufferu, než může skončit. Nemáme zaručeno, že skončí v konečném čase.
- *neblokující* – čekání po odeslání nebo přijetí není možné.

34. Explain how polling can be used to achieve non blocking message reception.

Neustále se odesílatele ptám, jestli už něco poslal => Mohu mít tedy blokující proceduru, ale ta je zablokovaná jenom po dobu co se ptám. Potom vykonávám něco jiného a po chvilce se opět zeptám, jestli už je zpráva pro mne připravena.

35. Explain how callbacks can be used to achieve non blocking message reception.

Ve zprávě, kterou odešlu, specifikuji svou funkci nebo proceduru: „Tady máš adresu funkce, kterou zavolej, až něco přijde.“ Jakmile „něco přijde“, zavolá mi přes RPC mojí proceduru.

36. Explain when a synchronous message sending blocks the sender.

- Odesílatel odešle zprávu a ta nedorazí příjemci.

37. Explain when an asynchronous message sending blocks the sender.

- odesílatel pošle zprávu, ale ta (vinou nějaké chyby) odesílatele ani neopustí.

38. Propose a blocking interface through which a process can send a message to another process. Use synchronous message passing with direct addressing.

```
MPI_SEND(buf, count, datatype, dest, tag)
- IN buf initial address of send buffer (choice)
- IN count number of elements in send buffer (nonnegative integer)
- IN datatype datatype of each send buffer element (handle)
- IN dest rank of destination (integer)
- IN tag message tag (integer)
```

39. Propose a blocking interface through which a process can send a message to another process. Use asynchronous message passing with indirect addressing.

```
MPI_SEND(length, msg, tag, comm)
- IN comm communicator (handle)
```

Process synchronization

40. Explain what is a race condition.

Chyba při komunikaci procesů, která je časově závislá; závisí na přeplánování. Komunikaci může představovat i psaní nebo čtení z/do sdílené proměnné.

41. Explain what is a critical section.

Místo v kódu, kde se vykonává kritická operace a kde může nastat race condition.

42. Explain the conditions under which a simple I++ code fragment on an integer variable can lead to a race condition when executed by multiple threads.

- Předpokládám neatomickou inkrementaci, potom lze I++ v assebleru zapsat: hodnotu načtu | přičtu 1 | zapíšu.
- Když dojde v místě označeném jako | k přeplánování a někdo bude pracovat s proměnnou I a změní jí, došlo k race condition.

43. Explain the conditions under which omitting the volatile declaration on an integer variable can lead to a race condition when the variable is accessed by multiple threads.

Vynechání volatile by mohlo vadit například u stavové proměnné zámku. Pokud by se překladač, nedej Bože, rozhodl práci s ní optimalizovat a nechat její hodnotu jen v registru, protože se stejně za chvíli zase bude měnit, tak zámky přestanou fungovat. Volatile tenhle druh optimalizace zakáže.

44. Describe the Mutual Exclusion synchronization task. Draw a Petri net illustrating the synchronization task and present an example of the task in a parallel application.

Úkolem je, aby dva procesy, které vykonávají svůj kód souběžně, nevstupovaly do svých kritických sekcí.
příklad: inkrementace nějakého čítače (neuvažuji atomickou inkrementaci).

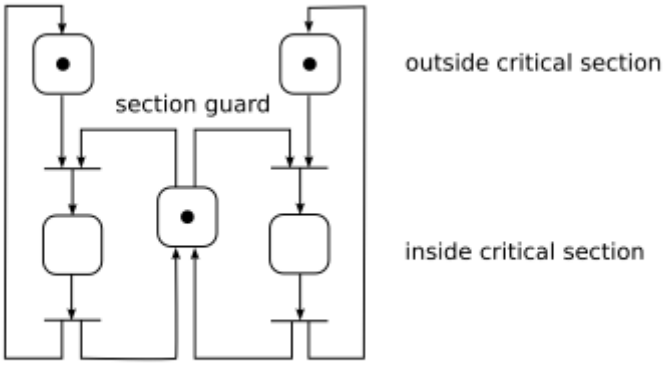


Figure 2-33. Mutual Exclusion Petri Net

45. Describe the Rendez Vous synchronization task. Draw a Petri net illustrating the synchronization task and present an example of the task in a parallel application.

Úkolem je, aby se dva procesy navzájem nepředběhly, tedy aby se v určitý okamžik oba setkaly na jednom místě a teprve potom oba najednou pokračovaly v běhu.
příklad: předání jednoduché proměnné mezi dvěma procesy. Randezvous dostane jako parametry tag a hodnotu. Tag je typicky adresa v paměti sdílené mezi oběma procesy. Volání randezvous způsobí uspání volajícího, dokud nenastane volání randezvous se stejným tagem a jiným procesem. Poté si hodnotu vymění a oba jsou probuzeni.

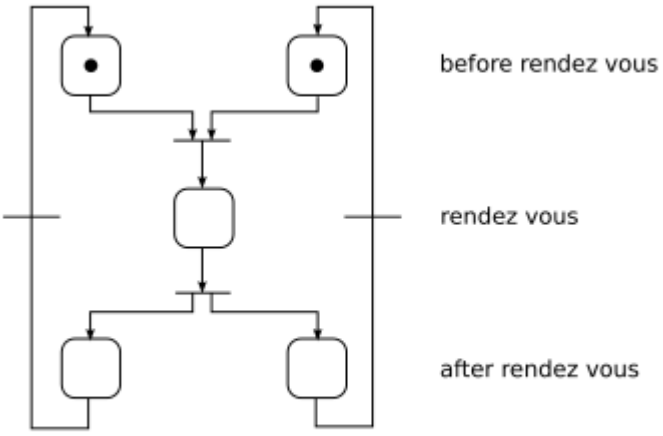


Figure 2-34. Rendez Vous Petri Net

46. Describe the Producer And Consumer synchronization task. Draw a Petri net illustrating the synchronization task and present an example of the task in a parallel application.

Úkolem je zajistit, aby buffer, který využívají dva procesy (producent, konzument) ani nepřetekl, ani nepodtekl, tedy producent nevyrábí, když nemá kam výrobek uložit a konzument nebude bouchat na dveře prázdného skladu :)
příklad: server s pevnou frontou zpráv.

47. Describe the Readers And Writers synchronization task. Draw a Petri net illustrating the synchronization task and present an example of the task in a parallel application.

Úkolem je zajistit možnost více čtenářům číst sdílenou proměnnou (paměť) a zamezit zapisovatelům, aby zapisoval v jednu chvíli více než jeden. Když se zapisuje, nikdo v tu chvíli nesmí číst.

48. Describe the Dining Philosophers synchronization task. Draw a Petri net illustrating the synchronization task and present an example of the task in a parallel application.

Úkolem je umožnit několika procesům exkluzivně sdílet prostředky.
příklad: několik procesů se uchází o exkluzivní vlastnictví prostředků, kdo má resource, tak si jej zamkne. Chce-li stejný zdroj jiný proces, uspí se, dokud není prostředek znovu odemčen.

49. Explain how a deadlock can occur in the Dining Philosophers synchronization task. Propose a modification of the synchronization task that will remove the possibility of the deadlock.

Například proces potřebuje k psaní dva soubory a takových procesů je více. Jestliže si každý proces zamkne jeden soubor, nikdo nepoběží (deadlock). Modeluje se exkluzivní potřeba více zdrojů.
rešení 1: když zamkne, nebude mít možnost psát a tak odemkne a to budou dělat všichni, nikdo zase nic nenapíše, ale systém bude maximálně vytížen. Řeší se sice deadlock, ale nastává starvation.
řešení 2: na začátku mají všichni jednu špinavou vidličku. Když chce filosof jíst, řekne si těm vedle sebe o vidličku. Filosof vedle mu jí podá, pouze pokud jí má a je špinavá (tedy před chvílí s ní jedl), protože je slušný, cestou jí vyčistí. Kdyby měl čistou, nedá mu jí. Po dojedení jsou obě vidličky špinavé. Jedná se o velmi efektivní algoritmus!

50. Explain the difference between active and passive waiting. Describe when active waiting and when passive waiting is more suitable.

aktivní čekání: neustále se testuje nějaká podmínka, tím se vlastně čeká. Hodí se pro krátké časy čekání, nebo pokud není k dispozici vhodný signál na ukončení pasivního čekání.
–: Spotřebovává se čas procesoru.
+: Krátká doba, neobtěžuje OS, rychlejší.
pasivní čekání: proces, který čeká na nějakou podmínku, je uspán. Probouzí se podnětem z vnějšku, tedy pokud v jiném procesu nastala sledovaná událost. Hodí se, pokud se čeká dlouhý čas, tedy nehraje roli zpoždění způsobené uspáním a opětovným probuzením procesu. Musí být také k dispozici vhodný signál na ukončení pasivního čekání.

51. Present a trivial solution to the mutual exclusion problem without considering liveness and fairness. Use active waiting over a shared variable. Explain the requirements that your solution has on the operations that access the shared variable.

- LIVENESS – Nenastane deadlock
- FAIRNESS – Zaručuje spravedlnost.

```
while (true)
{
  // Indicate the intent to enter the critical section
  bIWantToEnter = true;
  // Enter the critical section if the other
  // process does not indicate the same intent
  if (!bHeWantsToEnter) break;
  // Back off to give the other process
  // a chance and continue the active
  // waiting cycle
  bIWantToEnter = false;
}

// Code of critical section comes here
...

bIWantToEnter = false;
```

52. Present a solution to the mutual exclusion problem of two processes including liveness and fairness. Use active waiting over a shared variable. Explain the requirements that your solution has on the operations that access the shared variable.

Princip: na řadě je vždy ten druhý, když nechce, jdu já – Dekkerův algoritmus

```
// Indicate the intent to enter the critical section
bIWantToEnter = true;
while (bHeWantsToEnter)
{
  // If the other process indicates the same intent and
  // it is not our turn, back off to give the other
  // process a chance
  if (iWhoseTurn != MY_TURN)
  {
    bIWantToEnter = false;
    while (iWhoseTurn != MY_TURN) { }
    bIWantToEnter = true;
  }
}

// Code of critical section comes here
...

iWhoseTurn = HIS_TURN;
bIWantToEnter = false;
```

Požadavky na sdílenou proměnnou: spravedlivá implementace pro N procesů musí používat minimálně N sdílených proměnných, které lze číst a zapisovat, nebo N/2 proměnných, které lze číst, zapisovat a měnit pomocí AtomicSwap:

```
while (AtomicSwap (bCriticalSectionBusy, true))
{
// Active waiting cycle until the
// value of the bCriticalSectionBusy
// variable has changed from false to true
}
// Code of critical section comes here
...
bCriticalSectionBusy = false;
```

53. Describe the priority inversion problem.

Priority inversion je situace, kdy procesy s vyšší prioritou čekají na něco, co vlastní procesy s nižší prioritou. V nejhorším případě může priority inversion vést i k deadlocku.

54. Explain how priority inheritance can solve the priority inversion problem for simple synchronization primitives.

Z principu se podpora priority inheritance zdá být jednoduchá. V okamžiku, kdy proces začne na něco čekat, se jeho priorita propůjčí procesu vlastnícímu to, na co se čeká. Problém je v tom, že tohle funguje dobře u zámků, které mají jednoho vlastníka, ale u semaforů nebo condition variables se už nedá zjistit, kdo vlastně bude ten proces, který vázaný prostředek uvolní, a komu se tedy má priorita půjčit.

55. Present a formal definition of a deadlock.

Podmínka, na níž je blokováno dva nebo více procesů, každý čeká na uvolnění zámku vlastněným tím druhým procesem.

56. Present a formal definition of starvation.

Běžícímu procesu jsou neustále nedostupné potřebné prostředky, bez kterých nemůže dokončit úlohu. Nastává v případě, že proces drží prostředky jinému a nechce nebo není schopen se jich vzdát. Druhý proces tedy běží, žádá o prostředky, ale doběhnout nemůže.

57. Present a formal definition of a wait free algorithm.

The ability to provide bounds on waiting is essential in many synchronization problems. A solution to a synchronization problem is said to be
wait free when *every* process is guaranteed to finish in a finite number of its own steps,
lock free when *some* process is guaranteed to perform operations in a finite number of its own steps,and
obstruction free when every process is guaranteed to perform operations in a finite number of its own steps provided other processes take no steps.
Wikipedia says:
In contrast to algorithms that protect access to shared data with locks, lock-free and wait-free algorithms are specially designed to allow multiple threads to read and write shared data concurrently without corrupting it. "**Lock-free**" refers to the fact that a thread cannot lock up: every step it takes brings progress to the system. This means that no synchronization primitives such as mutexes or semaphores can be involved, as a lock-holding thread can prevent global progress if it is switched out. "**Wait-free**" refers to the fact that a thread can complete any operation in a finite number of steps, regardless of the actions of other threads. All wait-free algorithms are lock-free, but the reverse is not necessarily true.

58. Describe the interface of a lock and the sematics of its methods.

```
void mutex_init (struct mutex * mtx)
Funkce inicializuje mutex do stavu odemceno.
* void mutex_destroy (struct mutex * mtx)
Funkce provede uklid ridici struktury mutexu. Pokud je rusen mutex, na
kterem jsou jeste zablokovana nejaka vlakna, funkce vyvola panic.
* void mutex_lock (struct mutex * mtx)
* int mutex_lock_timeout (struct mutex * mtx, const unsigned int usec)
Funkce se pokusi zamknout mutex, pokud to neni mozne, zablokuje vlakno na
mutexu. Varianta _timeout ceka na zamknuti mutexu nejdele usec mikrosekund.
Pokud se behem teto doby podari mutex zamknout, vraci EOK, jinak ETIMEDOUT.
Pokud je casovy limit 0, k zablokovani vlakna nedochazi.
* void mutex_unlock (struct mutex * mtx)
Funkce odemkne zadany mutex a vzbudi vlakna, ktera byla zablokovana pri
zamykani mutexu. Pokud je symbol DEBUG_MUTEX >= 1, bude implementace mutexu
hlidat, zda je odemykan vlaknem, ktere ho puvodne zamknulo.
```

59. Describe the interface of a read-write lock and the sematics of its methods.

```
- rwlock_init – inicializuje počet čtenářů, vytváří mutex a vrací pointer
na rwlock
- rwlock_write_lock – vstup písaře do kritické sekce, uzamčeno pro zápis
- rwlock_read_lock – inkrementuje počet čtenářů, vstup do kritické sekce
- rwlock_write_unlock – oznámení ukončení psaní
- rwlock_read_unlock – odchod jednoho čtenáře z kritické sekce
- možné ještě trylock a lock_timeout
```

60. Explain when a lock is a spin lock.

Zámek čeká na uvolnění aktivně, spinning označuje opakované testování přístupu.

61. Explain when a lock is a recursive lock.

Zámek lze zamknout vícekrát stejným vlastníkem. Zámek je odemčen, až když se odemkne tolikrát, kolikrát byl do té doby zamčen.

62. Explain why Windows offer Mutex and CriticalSection as two implementations of a lock.

- *Critical sections* ve Windows NT si pamatují vlastníka, je možné je zamknout jedním vláknem několikrát, jsou poměrně rychlé, ale fungují pouze v rámci jednoho procesu.
- *Mutexy* – pro synchronizaci mezi procesy. Mají také vlastníka a lock count.

63. Implement a solution for the mutual exclusion synchronization problem using locks.

Oba sdílí jeden zámek, před vstupem do kritické sekce proces zámek zamkne, pro ukončení kritické sekce se zámek zase odemkne

64. Describe the interface of a semaphore and the sematics of its methods.

```
* void sem_init (struct semaphore * sem, const int value)
Funkce inicializuje semafor a nastavi ho na zadanou hodnotu.
* void sem_destroy (struct semaphore * sem)
Funkce provede uklid ridici struktury semaforu. Pokud je rusen semafor, na
kterem jsou jeste zablokovany nejake thready, funkce vyvola panic.
* int sem_get_value (struct semaphore * sem)
Funkce vrati hodnotu semaforu >= 0.
* void sem_up (struct semaphore * sem)
Funkce inkrementuje hodnotu semaforu a pripadne odblokuje jedno z vlaken na
nem zablokovanych.
* void sem_down (struct semaphore * sem)
* void sem_down_timeout (struct semaphore * sem, const unsigned int usec)
Funkce dekrementuje hodnotu semaforu, pokud je to mozne, jinak vlakno na
semaforu zablokuje. Varianta _timeout ceka na dekrementovani hodnoty
semaforu nejdele usec mikrosekund. Pokud se behem teto doby podari semafor
dekrementovat, vraci EOK, jinak ETIMEDOUT. Pokud je casovy limit 0, k
zablokovani vlakna nedochazi.
```

65. Implement a solution for the producer and consumer synchronization problem over a cyclic buffer using semaphores.

```
semaphore mutex = 1; //slouží k atomickému přístupu do skladu
semaphore empty = N;
semaphore full = 0;
PRODUCER:
produce_item(&item)
down(&empty)
down(&mutex)
enter_item(item)
up(&mutex)
up(&full)

CONSUMER:
down(full)
down(mutex)
remove_item(&item)
up(&mutex)
up(&empty)
consume_item(item)
```

66. Describe the interface of a condition variable and the sematics of its methods.

Conditional variables je nejobecnější synchronizační primitivum. Pokud vlákno čeká na condition variable, je blokováno, dokud jiné vlákno nesignalizuje condition variable. Condition variable (na rozdíl od semaforu) není čítač, vlákno na ni musí čekat před tím, než je signalizována. Jinak je signál ztracen a vlákno čeká na další. Je tu ale jeden problém: race condition mezi testováním podmínky a čekáním na condition variable. Např. vlákno zjistí, že podmínka neplatí. Plánovač přepne vlákna, podmínka platit začne a signalizuje se condition variable. Signál je ztracen, plánovač přepne vlákno zpět a to začne čekat na condition variable, které se možná nedočká. Potřebujeme zamknout mutexem podmínku i condition variable. Každá condition variable proto musí být použita spolu s mutexem:

- vlákno zamkne mutex a otestuje podmínku
- pokud je podmínka platí, odemkne mutex a vykoná činnost
- pokud ne, *atomicky* odemkne mutex a čeká na condition variable

```
* void cond_init (struct cond * cvar)
Funkce inicializuje ridici strukturu condition variable.
* void cond_destroy (struct cond * cvar)
Funkce provede uklid struktury condition variable. Pokud je rusena
condition variable, na ktere jsou jeste zablokovany nejake thready, funkce
vyvola panic.
* void cond_signal (struct cond * cvar)
* void cond_broadcast (struct cond * cvar)
Funkce odblokuje jedno resp. vsechna vlakna cekajici na condition variable.
* void cond_wait (struct cond * cvar, struct * mtx)
* int cond_wait_timeout ( struct cond * cvar, struct mutex * mtx, const
unsigned int usec)
Funkce atomicky odemkne mutex a zablokuje volajici vlakno na condition
variable. Po odblokovani opet zamkne mutex a vrati rizeni volajicimu.
Varianta _timeout zablokuje vlakno nejdele na usec mikrosekund, resp. na
dobu odpovidajici limitu a dobu nutnou k opetovnemu zamceni mutexu. Pokud
behem teto doby dojde k odblokovani vlakna, vraci EOK, jinak ETIMEDOUT.
Pokud je casovy limit 0, k zablokovani vlakna nedochazi, ale pred opetovnym
zamcenim mutexu se vlakno jednou vzda procesoru.
```

67. Explain what is a monitor as a synchronization tool and what methods it provides.

Monitor je synchronizační primitivum, pro řízení přístupu ke sdíleným prostředkům. Jeho zvláštností je, že jde o speciální konstrukci programovacího jazyka (musí ho tedy implementovat překladač), typicky implementovanou pomocí jiného synchronizačního primitiva. Výhodou monitoru oproti jiným primitivům je jeho vysokoúrovňovost – snadněji se používá a je bezpečnější. Při jeho použití je méně pravděpodobné, že programátor udělá chybu.

Monitor se skládá z dat, ke kterým je potřeba řídit přístup, a množiny funkcí, které nad těmito daty operují. Monitor se podobá třídě z OOP. Odlišností je to, že překladač doplní monitor o zámek, díky němuž se dosáhne vzájemného vyloučení – v jednu chvíli může být uvnitř monitoru jen jeden proces.

Když chce proces vstoupit do monitoru (tj. zavolat jeho funkci), musí nejdříve získat zámek. Pokud zámek v tu chvíli drží někdo jiný, tak se proces zablokuje a čeká, dokud se zámek neuvolní (tj. dokud jiný proces neopustí monitor nebo nezačne čekat na podmíněnou proměnnou). Celý proces zamykání je pro programátora transparentní. V programu se funkce monitoru volají stejně jako ostatní funkce. Kód, který provádí zamykání a odemykání, vygeneruje překladač.

V prostředí .NET (a tedy např. programovací jazyk C#) se používá prakticky stejný princip: každý objekt má svůj monitor, obdobou klíčového slova synchronized je v C# klíčové slovo lock. Navíc však .NET poskytuje i nízkoúrovňovější přístup k monitorům pomocí třídy Monitor a jejích metod (např. Enter a Exit), které dovolují o něco flexibilnější (ale „nebezpečnější“) řízení vzájemného vyloučení. Operace wait, notify a notifyAll jsou v .NETu podporovány taktéž prostřednictvím metod třídy Monitor (konkrétně Wait, Pulse, PulseAll).

Memory management among processes

68. Internal fragmentation leads to poor utilization of memory that is marked as used by the operating system. Explain how internal fragmentation occurs and how it can be dealt with.

Interní fragmentace nastává, pokud uživatel dostane větší blok paměti, než si objednal/využívá. To se stává například u heap alokátorů, které umí přidělovat bloky jen o velikostech v násobcích nějaké konstanty. Řešení zas tak moc asi není, jenom přidělovat bloky co nejbližších velikostí jako požadované. Uživatel typicky neví, že má víc paměti, než chtěl, tak s tím moc neudělá.

69. External fragmentation leads to poor utilization of memory that is marked as free by the operating system. Explain how external fragmentation occurs and how it can be dealt with.

Externí fragmentace spočívá v tom, že v paměti jsou mezi velkými bloky roztroušeny malé, které jsou ale tak malé, že je nikdo nechce. V součtu má tedy systém stále hafo volné paměti, ta je ale rozdrobena do nepoužitelných kousků. Stává se to například zase u těch heap alokátorů, když se hodně alokuje/uvolňuje. Řešení je například bloky setřepat a tím volné místo spojit, to ale jde jenom tehdy, pokud v kódu nejsou absolutní adresy.

70. Explain how memory virtualization through paging works and what kind of hardware and operating system support it requires.

Procesy nepracují přímo s fyzickou pamětí počítače. Místo toho se jejich virtuální adresní prostor mapuje na fyzický. Pro tyto účely jsou jak virtuální tak fyzický prostor rozděleny do bloků relativně malé pevné velikosti (stránek). Fyzické i virtuální stránky jsou stejně velké. Když proces přistupuje na nějakou adresu, tak ta se nejprve přeloží na fyzickou a s tou se teprve pracuje. Problém je právě v tom překladu a na to je potřeba podpora HW a OS. HW musí vědět, že adresy jsou virtuální a musí mít nějakou jednotku, která se stará o překlady. OS musí dát HW k dispozici data k překladu (stránkovací tabulky). Musí tyto tabulky vyplňovat a udržovat, přepínat je při přepnutí paměťového kontextu a případně taky stránky swapovat.

71. At the level of individual address bits and entry flags, describe the process of obtaining physical address from virtual address on a processor that only has the Translation Lookaside Buffer. Explain the role of the operating system in this process.

Při překladu adresy se vezme počáteční úsek bitů adresy a zkusí se vyhledat v TLB. Délka tohoto úseku závisí na velikosti stránky a samozřejmě na délce adresy (při 32bit adresách, 4kB stránkách je to 20 bitů). Pokud se mapování v TLB najde, tak se k číslu rámce z TLB přičte zbytek bitů adresy, které tvoří offset do stránky a to dá výslednou fyzickou adresu. Bity, které typicky ještě ovlivňují vyhledávání:

- Global bit - mapování je globální, platí ve všech adresních prostorech ASID
- Address Space Identifier, pokud není nastaven global bit, při hledání mapování se porovnává ještě identifikátor aktuálního adresního prostoru
- Valid bit Indikuje, zda je mapování platné
- Present bit Indikuje, zda je stránka v paměti, nebo je odswapovaná.

Role operačního systému je udržovat TLB platnou a naplňovat jí, pokud v TLB není nalezeno mapování.

72. At the level of individual address bits and entry flags, describe the process of obtaining physical address from virtual address on a processor that supports multilevel page tables. Explain the role of an operating system in this process.

Při n úrovňových stránkovacích tabulkách se virtuální adresa rozdělí na n+1 úseků, z nichž prvních n (od nejvyšších řádů) slouží jako indexy do stránkovacích tabulek příslušné úrovně a poslední úsek je opět offset do stránky. První úsek slouží jako index do adresáře stránkovacích tabulek, který jediný musí být vždy namapovaný. Flagy jsou podobné jako u TLB. Role systému je vyplňování a udržování stránkovacích tabulek a taky přepnutí adresáře stránek při přepínání paměťového kontextu.

73. Explain the relation between the page size and the size of the information describing the mapping of virtual addresses to physical memory and list the advantages and disadvantages associated with using smaller and larger page sizes.

Velikost stránky je rozhodující pro velikost tabulky pro překlad (VA->FA). Pro adresový prostor 32 bitů určuje 12 bitů offset do stránky, tedy stránka bude mít 4KB. Zbývá 20 bitů na adresu stránky, tedy 2^20 položek. Bude-li mít položka velikost 4 byte, potřebujeme tabulku 4MB.
Velké stránky nám ukrojí moc velkou část na offset, naopak malé stránky neúnosně zvětšují překladovou tabulku.

74. List the advantages and disadvantages of using multilevel page table as a data structure for storing the mapping of virtual to physical memory.

+: Není třeba rezervovat prostoru pro tabulku stránek celé virtuální paměti, stačí pro použitou část, části tabulky mohou být stránkovány, tabulky v první úrovni mohou mít záznam „not present“, čímž se šetří místo pro tabulky na nižších úrovních, na které by ten záznam ukazoval
–: zvyšuje se počet přístupů do paměti, zpomaluje procesor.

75. List the advantages and disadvantages of inverse page table as a data structure for storing the mapping of virtual to physical memory.

Na 64. Bitových CPU kde Fyzická paměť << Virtuální Paměť. „Rámcovací“ tabulky se indexují fyzickými adresami. Postup: Hašuji část VA, získám index do „Rámcovací“ tabulky, která je adresovaná číslem rámce. Obsahuje část VA + odkaz na další záznam, který se semka taky nahašoval. Až to celé prolezu, kouknu se na index, ve kterém jsem, posunu a přičtu offset.
+: Je to rychlé, velikost tabulky závisí na velikosti fyzické, nikoli virtuální paměti.
–: Špatně se prohledávají hašováním, řešit kolize by bylo nákladné a tak je třeba ještě dodatečných datových struktrur, které využívá OS.

76. Explain what a Translation Lookaside Buffer is used for.

Cache pro překlad virtuálních na fyzické adresy, první místo, kam se procesor dívá, zdali existuje mapování. Pokud ne, zkoumá překladové tabulky nebo dráždí OS.

77. Describe the hardware realization of a Translation Lookaside Buffer and explain the principle and advantages of limited associativity.

Asociativní paměť, což je paměť, která hledá podle obsahu.
Položky TLB jsou rozděleny do množin (bank) s velikostí N, kde N udává číslo asociativity. Samotné asociativní prohledávání spočívá v HW realizaci pomocí komparátorů, paměť je tedy asociativní pouze v rámci každé množiny = limited cache asociativity.
+: Je to levnější než fully asociativity.

78. How does the switching of process context influence the contents of the Translation Lookaside Buffer ? Describe ways to minimize the influence.

Při přepnutí kontextu je třeba invalidovat i TLB (instrukce TLB_INVALIDATE) - většinou se z tabulek vše vyhodí a poté se znovu naplní. Na multiprocesorech má každý CPU svou TLB, musí se tedy vyprázdnit všechny (musí se dát vědět všem virtuálním adresovým prostorům, aby nikde nezůstaly neplatné virtuální adresy).
[Na MIPSu, kde můžeme mít až 255 ASIdů a jen 48 záznamů se TLB invalidovat nemusí, pouze se musí hlídat, když se odstraní všechny záznamy s daným ASID, tak aby se invalidoval ASID u VAPu.]

79. Provide at least two criteria that can be used to evaluate the performance of a page replacement algorithm.

- kolik informací je potřeba skladovat, aby algoritmus vůbec mohl rozumně běžet.
- počty výpadků stránek za běhu aplikace.
- rychlost algoritmu – dlouhé hledání volných stránek je mi na tužku.

80. Explain the principle of the First In First Out page replacement algorithm and evaluate the feasibility of its implementation on contemporary hardware along with its advantages and disadvantages.

princip: vyhodí se ta stránka, která přišla před nejdelší dobou, je nejdéle v paměti.
realizace: velmi dobrá, drží se pointer na nejstarší stránku a po jejím vyhození se posune.
+: je rychlý, snadno realizovatelný.
–: Beladyho anomálie (012301401234), nezohledňuje využívání stránek procesem (například stránka s velice často využívanými proměnnými bude určitě vyhozena a bude se muset znovu zavádět).

81. Explain the principle of the Not Recently Used page replacement algorithm and evaluate the feasibility of its implementation on contemporary hardware along with its advantages and disadvantages.

princip: na kterou stránku se nikdo dlouho nedíval ani do ní nepsal, tu asi nikdo nebude potřebovat i v nejbližší době – princip lokality, algoritmus předpokládá, že referencovaná stránka je důležitější než změněná.
realizace: množiny stránek rozdělené do čtyř kategorií podle nastavených Access a Dirty bitů (A0D0, A0D1, A1D0, A1D1), vyhazují se vždy stránky z nejnižší kategorie.
Podmínka: HW periodicky schazuje Access bit, například po určitém počtu přerušení od timeru, A i D bity se nahazují/shazují hardwarově, na čtyři skupiny si stránky dělí OS.
+: asi žádné velké, strategie se moc nepoužívá.

82. Explain the principle of the Least Recently Used page replacement algorithm and evaluate the feasibility of its implementation on contemporary hardware along with its advantages and disadvantages.

princip: stránky používané před krátkým okamžikem budou asi opět brzy použity, budeme si tedy pamatovat dobu, kdy byly stránky použity naposledy a vyhazujeme ty s největší hodnotou čítače
realizace: v HW – 64. bitový čítač inkrementovaný po každé instrukci - moc se nepoužívá.
SW – linked list (nepoužívá se, moc velká režie)
další možnost – matice n krát n podle počtu rámců, při použití rámce k, nastavit řádek k na samé jedničky a vynulovat sloupec k. Při výpadku se pak vybírá rámec s nejnižší binární hodnotou.
+: dá se statisticky zpracovat, dokázalo se, že LRU nezpůsobí nikdy víc než n-krát víc výpadků oproti optimálnímu algoritmu.
–: známé případy, kdy algoritmus selhává (například aplikace, která bude procházet N+1 stránek způsobí výpadek při každém přístupu).

83. Explain the principle of the Least Frequently Used page replacement algorithm and evaluate the feasibility of its implementation on contemporary hardware along with its advantages and disadvantages.

princip: stránka s nejmenší frekvencí přístupů za poslední dobu se vyhodí – SW simulace LRU. SW čítač ke každé stránce.
realizace: v současném HW je velmi bídná.
–: Špatné chování algoritmu při změně vytížení stránek (například po startu systému kdy už startovací stránky nebudou potřeba, ale nevyhazují se, protože byly intenzivně využívány).
+: samy o sobě nejsou, strategie různých algoritmů se musí kombinovat.

84. Explain what is a working set of a process.

Definicí může být více:

- Množina stránek ve fyzické paměti, které patří určitému procesu, mění se s během procesu.
- Množina stránek, které proces potřebuje, aby mohla nějakou dobu běžet bez výpadku stránky.

Pokud je working set příliš velký, může běžet zároveň méně procesů, když příliš malý, musí se častěji spouštět swapování.

85. Explain what is a locality of reference and how it can be exploited to design or enhance a page replacement algorithm.

Předpokládá se, že se program v určitých částech bude stále „motat na místě jako vítr v bedně“, tedy je vhodné nejčastěji přistupované stránky za poslední dobu nebo naposledy navštěvované stránky nevyhazovat.

86. Explain what is thrashing and what causes it. Describe what can the operating system do to prevent thrashing and how can the system detect it.

V případě že se working sets běžících procesů nevejdou do paměti, jedna stránka se vyhodí a začne se swapovat, jenže proces, jemuž byla stránka vyhozena, chce běžet, a tak se začne swapovat stránka zpátky a tak pořád dokolečka – nakonec se jen swapuje a neudělá se nic.
řešení: vyswapuje se celý jeden proces (obětní beránek) a nepoběží vůbec, zvýšení RAM.
detekce: ponechá se rezerva, přes kterou už nebudu povolovat ukládání do paměti. Jakmile se překročí, začne se swapovat „dopředu".

87. Explain the concept of memory mapped files.

Díky virtuální paměti máme možnost namapovat velký externí soubor do paměti a přistupovat k němu, jako by byl v paměti celý, což urychluje to práci se soubory, protože se nepřistupuje k externím pamětem. Podstatné je, že pro práci s MMF není třeba volat systém, ale používají se běžné instrukce pro práci s pamětí.

88. Explain the priciple of the copy-on-write mechanism and provide an example of its application in an operating system.

Pokud chce více procesů sdílet stejná data (například kód), OS je do paměti namapuje pouze jednou (data) a označí je copy on write (vlastně read-only a jakmile chce na stránku někdo psát, vyvolá se vyjímka, OS se ujme řízení a stránka je okopírována jinam do paměti, kde se do ní zapíše).

Allocation within a process

89. Identify where the following function relies on the virtual address space to store executable code, static data, heap and stack.

```
void *SafeAlloc (size_t iSize)
{
  void *pResult = malloc (iSize);
  if (pResult == NULL)
  {
    printf ("Failed to allocate %z bytes.\n", iSize);
    exit (ENOMEM);
  }
  return (pResult);
}
```

Parametry fuknce a lokální proměnné funkce na zásobník; malloc na haldu, kód a statický řetězec do kódového segmentu

90. List four distinct types of content that reside in a virtual address space of a typical process .

- executable code – strojový kód v instrukcích procesoru
- static data – staticky alokované proměnné
- heap – halda, dynamicky alokovaná data
- stack – zásobník, pro ukládání návratových adres, argumentů procedur, registrů a lokálně alokovaných dat

Kod a static data jsou neměnné a jejich poloha je determinovana překladačem. Stack a heap jsou dynamicke a rostou proti sobě [viz semestrálka].

91. Explain the advantages of segmentation over flat virtual memory address space.

HW podpora pro relokaci. SEGMENT:OFFSET (OFFSET např 32bitů). Velikost každého segmentu může být individuální. Není třeba řešit rozložení heap X zásobník a jiné bloky paměti. Důležité je, že každý blok paměti (executable code, heap, stack…) bude jeden segment, což nám umožní i efektivně zjišťovat, zda například program nesahá „nad“ zásobník, při porovnávání adresy se segment registrem by se to zjistilo a proces který tak učinil, by byl zastřelen.

- No page faults -- whole segment is present at once -- only access violations need to be trapped
- No internal fragmentation -- segment size is customized to each process
- Easier to implement virtual OS per segment protection
- Snadnejsi sdileni dat (sdilim funkci/ ne naky stranky).

92. Explain why random placement of allocated blocks in the virtual address space of a process can contribute to improved security.

Protože potom útočník nemůže dělat žádné závěry o tom, kde bude blok v paměti uložen, a to mu znesnadňuje „buffer overflow attack“ (útok spočívá v zápisu hodnoty delší než je alokovaná vstupní proměnná, čímž se přepíše jiná hodnota na zásobníku, která leží pod vstupní proměnnou. Při návratu z procedure je návratový kód vyzvedávaný ze zásobníku jiný a může směřovat na sadu škodlivých, útočníkem podsunutýc instrukcí).
Náhodné rozmístění zásobníků, …

93. Explain what the processor stack is used for with typical compiled procedural programming languages.

Používá se pro uschovávání návratových adres, argumentů procedur a funkcí, dočasně uložených registrů procesoru a lokálně alokovaných proměnných. K přístupu na zásobník se používá stack pointer – regist procesoru.

94. For a contemporary processor, explain how the same machine instructions with the same arguments can access local variables and arguments of a procedure regardless of their absolute address in the virtual address space. Explain why this is important.

Příklad: Intel IA32
Instrukce *push* and *pop* – při volání procedury se sníží ESP register o velikost návratové adresy a ta se uloží. K adresování lokálních proměnnch a argumentů se využívá EBP – base pointer. Ten je po volání procedury uložen na zásobník a zároveň je do EBP uložena hodnota ESP. Na zásobníku se vyhradí místo pro proměnné tím, že se k SP přičte tolik byte, kolik lokální proměnné zabírají. Proměnné se uloží nad ESP, argumenty se uloží pod ESP.
Stack roste od větších k menším adresám, to má tu výhodu, že může být umístěn na konci paměťového prostoru. Nepoužívají se absolutní adresy do VAP, ale pouze relativní vzhledem EBP

95. Explain what is the function of a heap allocator.

Určuje umístění dynamicky alokovaných bloků paměti. Pracuje nad strukturou bloků s informacemi, které kusy haldy jsou prázdné a které volné.
Požadavky na alokátor – rychlost, malá režie, úspornost, a dodatečné fičury (rezise, garbage collector, debugging, zero-fill, align).

96. Explain why the implementation of the heap allocator for user processes usually resides in user space rather than kernel space.

Od heap alokátoru se typicky očekává rychlost a volá se často, proto by bylo neefektivní na každé jeho zavolání ještě přikládat celou režii kolem syscallů. To je podle mne hlavní důvod.

97. Design an interface of a heap allocator.

```
void * malloc(size_t size);
void free(void *ptr);
size_t get_size(void *ptr);
```

98. Explain the problems a heap allocator implementation must solve on multiprocessor hardware and sketch appropriate solutions.

Kdyby sloužil pro všechny procesory naráz, stal byse heap allocator úzkým hrdlem – při hledání volného místa by brzdil ostatní procesory.
Řešení: Hierarchický alokátor, který si udržuje „local free block pools“, tedy nabídku pro jednotlivé procesory s tím, že když na vrcholu hierarchie dojde místo, přelévají se z local do global free block poolu.
Multiprocesorové systémy mají u alokátoru podobné problémy jako plánovace nad ready frontou, tedy příliš mnoho soubehů je zpomaluje.

99. Explain the rationale behind the Buddy Allocator and describe the function of the allocation algorithm.

Princip: Půlení paměti – někdo požaduje blok paměť, kterou půlíme do té doby, než nemáme blok minimální možné velikosti, nebo velikosti větší než požadovaná paměť. Pro každou možnou velikost bloku jednu bitmapu. Při dealokaci se sousední volné bloky spojují.
+: Je jasně daná velikost bloků, přijatelná doba na alokování/dealokoání paměti
–: Vysoká fragmentace, která je daná pevnou sadou délek bloků

100. Explain the rationale behind the Slab Allocator and describe the function of the allocation algorithm.

princip: nejčastěji se alokují malá data, trik je v tom, že nejčastěji používané objekty jsou již připravené na stránkách, to snižuje režii odstraněním alokace těchto objektů, jedna stránka tak obsahuje řadu objektů, čímž se šetří paměť a snižuje se interní fragmentace.
Použití: solaris
terminologie: slab je kontejner na objekty a je tvořen několika stránkami, cache se skládají ze slabů, objekt je nejmenší jednotka alokace, nachází se ve slabu.

101. Describe what a heap allocator can do to reduce the overhead of the virtual memory manager.

Je výhodné, aby alokátor udržoval lokalitu, tedy aby nedávno alokované bloky umisťoval blízko sebe, protože budou pravděpodobně používány společně, takže v případě přestránkování nebude vyhazováno hodně stránek.
Alokátor nejprve zkouší exact fit, v případě, že to není možné, zkusí rozdělit volný blok, ze kterého se přidělovalo naposledy.
Statistiky overheadu pro konkrétní aplikace uvádejí 4% pro best fist, 7% pro first fit na FIFO seznamu volných bloku, 50% pro first fit na LIFO seznamu volných bloku, 60% pro buddy system.

102. Explain the function of a garbage collector.

Garbage collector SÁM uvolňuje paměť, kterou už žádný proces nevyužívá. Je třeba proto nějakým způsobem udržovat počet referencí na bloky spravované garbage collectorem.

103. Define precisely the conditions under which a memory block can be freed by a reference tracing garbage collector.

Dosažitelnost: pokud proces může sledovat retěz referení od staticky nebo lokálně alokovaných proměnných, tzv. ROOTS tak, že dojde ke každému bloku (jsou dosažitelné referencí z call stacku, nebo globálních proměnných); funguje tranzitivita.
Může se udělat bez kopírování (Mark&sweep), nebo s kopírováním, ale za běhu.

104. Describe the algorithm of a copying garbage collector.

Princip: Dosažitelné bloky kopíruji do nového heapu, nakonec starý heap nakonec smažu.
Časová složitost závisí na počtu živých objektů.
+: Vytváří se kompaktní heap.
–: Objekty se musí přestěhovat na jinou adresu.

105. Describe the algorithm of a mark and sweep garbage collector.

- Zastaví se provádění aktuálníhoprocesu
- Všechny bloky na haldě se na začátku označí jako garbage
- Procházím tranzitivně z ROOTS (data, co jsou z programu vždy vidět, jako globální & lokální proměnné) a kam přijdu, tam to označím jako používané.
- Až všechny položky z ROOTS tranzitivně projdu, odstraním zbylý garbage.

106. Describe the algorithm of a generational garbage collector. Assume knowledge of a basic copying garbage collector.

Pozorování1: Mnoho objektů se stane odpadem krátce po svém vzniku
Pozorování 2: Jen malé procento referencí ve „starších“ objektech ukazuje na objekty mladší
princip: Garbage collector si paměť dělí na části - generace, objekty se začínají vytvářet jako nejmladší generace, v případě že se zaplní, všechny dosažitelné objekty se přesunou (označí) do starší generace. Intervaly úklidů jsou různé pro různé generace, typicky velmi krátké pro nejmladší generaci, strategie úklidu může být různá pro různé generace.
Možným problémem jsou reference mezi generacemi; Musí se udržovat jejich seznam a jsou považovány za ROOTs.

Devices and device drivers

107. Popište obecnou architekturu ovladače zařízení a vysvětlete, jak tato architektura dovoluje zpracovávat současně asynchronní požadavky od hardware a synchronní požadavky od software.

Pro označení těchto dvou částí driveru se používají termíny bottom half (asynchronně volaná část driveru, která se stará převážně o požadavky hardware) a top half (synchronně volaná část driveru, která se stará převážně o požadavky software).
Tento problém se řeší použitím mechanismů, které dovolují naplánovat na později vykonání operací, které jsou součástí obsluhy přerušení od hardware (v Linuxu bottom half handlers a tasklets, ve Windows deferred procedure calls, v Solarisu pinned interrupt thread pools).

108. Popište obvyklý průběh obsluhy přerušení jako asynchronního požadavku na obsluhu hardware ovladačem zařízení. Průběh popište od okamžiku přerušení do okamžiku ukončení obsluhy. Předpokládejte obvyklou architekturu ovladače, kde spolu asynchronně a synchronně volané části ovladače komunikují přes sdílenou frontu požadavků. Každý krok popište tak, aby bylo zřejmé, kdo jsou jeho účastníci a kde získají informace potřebné pro vykonání daného kroku.

Dle státnic:

- Dokončí se aktuální instrukce
- OS se ujme řízení
- Uložení stavu CPU
- Analýza přerušení, vyvolání obsluhy
- Obsluha přerušení
- Obnovení stavu CPU

Osy poznámky:
Read/Write:

1) Přidej do fronty zařízení
2) Je disk aktivní? Pokud ano, tak 4), jinak 3).
3) Pošli požadavek disku
4) Sleep

Disk vyvolá IRQ -> IRQ Handler

A) AKNOWLEDGE
B) Najít požadavek ve frontě & WAKEUP
C) QUEUE Empty? Pokud není, tak 3) & 4).

Problémem je přístup do sdílených struktur. Deferred service – mezi A a B, odložení obsloužení.

- Soft irq – připojíme fuknci, která se zavolá až s povoleným přerušením. Používá se v reakci na síťové pakety, update časovačů.
- Tasklet – pro programátory driverů; zaručení, že nepoběží na víc procesů najednou. Volá se ze soft irq (nesmí se pasivně čekat). Implementují bottom half handler.
- Work queue – obdoba tasklet, ale může i pasivně čekat. Běží vždy v kontextu kernelového vlákna.
- Deffered procedure call – The interrupt service routine can register a deferred procedure call, which gets executed later. The decision when to execute a deferred procedure call depends on the importance of the call, the depth of the queue and the rate of the interrupts.

109. Popište obvyklý průběh systémového volání jako synchronního požadavku na obsluhu software ovladačem zařízení. Předpokládejte obvyklou architekturu ovladače, kde spolu asynchronně a synchronně volané části ovladače komunikují přes sdílenou frontu požadavků. Průběh popište od okamžiku volání do okamžiku ukončení obsluhy. Každý krok popište tak, aby bylo zřejmé, kdo jsou jeho účastníci a kde získají informace potřebné pro vykonání daného kroku.

Nejsem si úplně jistý, co tu chce všechno mít, ale tak nástřel: Uživatelské vlákno vyvolá obslužnou funkci přes rozhraní systémových volání (syscall, int 80h, …), tím vstoupí do režimu jádra a zde zavolá potřebnou rutinu na ovladači zařízení. Ta vloží požadavek do fronty požadavků (což případně může znamenat nějaké čekání na zámku, protože o frontu se perou jak další procesy, které něco chtějí, tak asynchronní obsluha zařízení, která z ní tahá požadavky po vyřízení předchozích). Poté se volající vlákno uspí a čeká na vyřízení svého požadavku. Po vyřízení ho asynchronní obsluha přerušení vzbudí, vlákno případně ještě někam přeuloží získaná data, vrátí se z rutiny ovladače a opustí režim jádra.

110. List the features that a hardware device that represent a bus typically provides.

Sběrnice = dráty + řadič (inteligence).
Parametry: Datová šířka, kapacita, rychlost. Technika přenosu po sběrnici je také DMA.
In computer architecture, a bus is a subsystem that transfers data or power between computer components inside a computer or between computers and typically is controlled by device driver software. Unlike a point-to-point connection, a bus can logically connect several peripherals over the same set of wires. Each bus defines its set of connectors to physically plug devices, cards or cables together.
- Although busses are not devices in the usual sense, devices that represent busses are sometimes available to control selected features of the busses. Most notable are features for bus configuration.
- Common features of busses include listing devices connected to the bus and drivers associated with the bus, matching drivers to devices, hotplugging devices, suspend- ing and resuming devices.

111. List the features that a hardware clock device typically provides.

Využití hodin, kalendář, plánování procesu v preemptivním multitaskingu, účtování strojového času, alarmy pro user procesy, watchdogs pro systém.
Principy hodinového hardware, odvození od sítě a od krystalového oscilátoru.
Možné funkce hardware, tedy samotný čítač, one time counter, periodic counter, kalendář.

112. List the features that a hardware keyboard device typically provides.

Klasický příklad character device. Klávesnice bez řadiče. Klávesnice s řadičem, překódování kláves, type ahead buffer, přepínání focusu.
Keyboards are designed for the input of text and characters and also to control the operation of a computer.

113. List the features that a hardware mouse device typically provides.

Computer-users usually utilize a mouse to control the motion of a cursor in two dimensions in a graphical user interface.

114. List the features that a hardware terminal device with command interface typically provides.

Standardy řídících příkazů. ANSI Escape Sequences třeba ESC [J (clear screen, 0 from cursor, 1 to cursor, 2 entire), ESC [; H (goto line and column), něco na barvy atd.

115. List the features that a hardware terminal device with memory mapped interface typically provides.

Popis terminálů s memory mapped interface, znakové a grafické displeje, práce s video RAM, akcelerátory, hardwarová podpora kurzoru, kreslení apod.

116. List the features that a hardware disk device typically provides.

Čtení a zápis z/na určitý sektor. Různé disky mají různou organizaci sektorů. Persistentní paměť, tj data se po vypnutí napájení uchovávají.

117. Explain the properties that a hardware disk interface must have to support hardware ordering of disk access requests.

Most versions of the ATA interface do not support issuing a new request to the disk before the previous request is completed, and therefore cannot implement any strategy to process the queue of requests. On the contrary, most versions of the SCSI and the SATA interfaces do support issuing a new request to the disk before the previous request is completed.

118. Describe at least three strategies for ordering disk access requests. Evaluate how the strategies optimize the total time to execute the requests.

- FIFO – požadavky na čtení/zápis se vyřizují v pořadí, ve kterém dorazily.
- The Shortest Seek First – vždy se vyřídí ten požadavek, který požaduje přístup na nejbližší pozici od aktuální pozice diskové hlavy (tím ale můžou strádat okraje disku).
- Výtah – Hlava se pohybuje směren dopředu a po cestě vyřizuje požadavky, na konci se buď okamžitě vrátí, nebo postupně vrací a přitom dále vyřizuje požadavky ve směru „jízdy“ (obousměrný).

119. Explain the role of a disk partition table.

[WIKIPEDIA CS:]Diskové oddíly slouží k rozdělení fyzického disku na logické oddíly, se kterými je možné nezávisle manipulovat. Laicky řečeno, po rozdělení pevného disku se jeden disk jeví jako několik nezávislých disků, které mohou být nejen různě naformátovány (tj. mít odlišnou logickou strukturu), ale mohou obsahovat dokonce i různé operační systémy.

Rozdělení fyzického disku na logické diskové oddíly bývá uvedeno v tzv. Partition Table (tabulka rozdělení disku), která se nachází na úplně prvním sektoru disku (tzv. Master boot record, hlavní spouštěcí záznam). Nejčastější verze této tabulky umožňuje pouze čtyři záznamy, ale v případě většího počtu diskových oddílů na jednom médiu je možné v hlavní (primární) tabulce odkázat na takzvaný rozšířený diskový oddíl (extendend partition), na jehož začátku se opět nachází MBR s další tabulkou, ve které je uvedeno rozdělení extended partition na další oddíly.

- Instalaci více operačních systémů na jeden disk.
- Obcházet omezení souborového systému. Mnoho souborových systémů má omezení velikosti logického oddílu, ať už ve smyslu limitu, nebo ve smyslu rostoucí neefektivity metadat (například u NTFS rozsáhlejší MFT tabulka způsobuje, že přístup k souborům na jednom velkém oddílu je pomalejší než na dvou menších).
- Použití různých souborových systémů nebo jejich parametrů pro různé účely.
- Ochranu izolací souborů. Logická poškození disku zasahují oddíly tak, že poškodí jeden oddíl a ostatní zůstanou nepoškozeny.

Tabulka s popisem rozdělení disku se nachází na jeho úplném začátku v takzvaném MBR (Master boot record, tj. prvních 512 bajtů na disku). V tomto prostoru se obvykle kromě tabulky nachází i zavaděč, který rozhoduje, ze kterého oddílu bude následně zaveden operační systém.

120. List the features that a network interface hardware device typically provides.

Lepší síťové karty umí DMA, nebo má nemapovanou paměť (variace na stejné téma).
Scatter/gather – Scatter – rozsekání a uložení na několik míst do paměti (TCP/IP hlavičky); Gather – „vysosnutí“ dat z několika míst paměti a slepení dohromady.
Nabízí služby typu odešli/přijmi data; Umí odpovídat na PING. Počítá kontrolní součty

121. Popište obvyklé rozhraní operačního systému pro přístup k souborům pomocí operací čtení a zápisu. Funkce rozhraní uveďte včetně argumentů a sémantiky.

```
int open (const char *pathname, int flags);
flags – RO/ RW, Append.
off_t lseek (int fildes, off_t offset, int whence);
whence – SEEK_BEGIN, SEEK_END, SEEK_XX (a pak se počítá s offsetem)?
ssize_t read (int fd, void *buf, size_t count);
ssize_t write (int fd, const void *buf, size_t count);
int close (int fd);
```

122. Vysvětlete, proč obvyklé rozhraní operačního systému pro přístup k souborům pomocí operací čtení a zápisu odděluje operace otevření a zavření souboru a operaci nastavení aktuální pozice v souboru od vlastních operací čtení a zápisu.

Důvody pro pětici operací jsou možnost odstranit při běžném způsobu práce se soubory opakování operací jako je nalezení pozice na disku ze jména souboru a pozice v souboru, možnost spojit otevírání souboru s dalšími operacemi jako je zamykání nebo kontrola oprávnění.
Důvodem pro dvojici operací je možnost implementovat file systém bezestavově, což přináší výhody v distribuovaných systémech.

123. Popište obvyklé rozhraní operačního systému pro přístup k souborům pomocí operací mapování do paměti. Funkce rozhraní uveďte včetně argumentů a sémantiky.

```
void *mmap (void *start, size_t length,int prot, int flags, int fd, off_t
offset);
ochrana dat PROT_READ, PROT_WRITE, PROT_EXEC
flags MAP_SHARED, MAP_PRIVATE (copy-on-write), MAP_FIXED
int munmap (void *start, size_t length);
void *mremap (void *old_address, size_t old_size, size_t new_size, unsigned
long flags);
```

124. Popište obvyklé rozhraní operačního systému pro práci s adresáři. Funkce rozhraní uveďte včetně argumentů a sémantiky.

```
DIR *opendir (const char *name);
int closedir (DIR *dir);
struct dirent *readdir (DIR *dir);
```
Základní operace na adresářích jsou otevření a zavření a čtení či prohledávání obsahu. Pro zápis obsahu jsou zvláštní funkce, které vytvářejí, přejmenovávají a mažou adresáře a soubory a nastavují jejich atributy, aby aplikace nemohly poškodit strukturu adresáře.

125. Popište obvyklé rozhraní operačního systému pro zamykání souborů. Funkce rozhraní uveďte včetně argumentů a sémantiky.

```
int flock(int fd, int operation);
pomoci fcntl v UNIXU, zada se offset, delka zamykaneho kusu a rezim
zamykani(shared, exclusive).
Musi se pamatovat seznam zámku na souboru
int fcntl(int fd,int cmd,struct flock *lock);
flock je struktura obsahující offset a délku, typ zámku.
```
Funkce jsou advisory, mandatory je nastavením dané nesmyslné kombinace atributů.

126. Vysvětlete rozdíl mezi advisory a mandatory zámky pro zamykání souborů. Vysvětlete, proč tyto druhy zámků existují.

Unix rozlišuje advisory a mandatory locking. Od začátku implementované jsou pouze advisory locks, totiž zámky, které se projeví pouze v případě, že se na ně proces zeptá. To samozřejmě není příliš bezpečné, a tak se doplnily ještě mandatory locks, které kontroluje kernel. Aby mandatory locks neblokovaly ve chvílích, kdy to stávající aplikace nečekaly, řeklo se, že budou automaticky nasazené na soubory s nastaveným group ID bitem a shozeným group execute bitem (což je nějaká hrozně nesmyslná kombinace).

127. Vysvětlete hlediska ovlivňující volbu velikosti bloků jako alokačních jednotek na disku.

Soubory se na disk ukládají po blocích. Disky jsou organizovány po sektorech.
Záleží na skutečné velikosti souborů, které chci na disku mít. Bloky souboru můžou být uloženy na disku prakticky náhodně, a protože je operace pro čtení z disku drahá, jsou z tohoto hlediska lepší větší bloky. Proti tomu ovšem stojí fragmentace, protože poté se nic menšího než blok nepřiděluje. Větší bloky mají také menší režii na správu.

128. Uveďte, jakými způsoby lze na disku ukládat informaci o blocích, ve kterých jsou umístěna data souborů. Jednotlivé způsoby ilustrujte na existujících systémech souborů a zhodnoťte.

Přechod z pásek na disky, první nápad se sekvenčním ukládáním souborů. Má to dvě výhody, totiž rychlost a malou režii. Nevýhodou je potřeba znát předem délku souboru a pochopitelně fragmentace.
První nápad jak tohle odstranit je udělat linked list. C64 měl tohle na floppy, nevýhody jsou zřejmé. Extrémně pomalý random access, velikost bloků není mocnina dvou, špatně se maže a tak.
Další modifikace je nechat linked list, ale vytáhnout ho z bloků a dát do tabulky. Typické řešení MS-DOSu. Nevýhodné to začne být, když se celá tahle tabulka nevejde do paměti, lidi napadlo mít jí po kouskách u souborů, výsledek je třeba CP/M nebo UNIX. Co dělat když je tahle tabulka moc velká, CP/M přidává lineárně další bloky, UNIX stromově větví I-nodes.
Souvislá alokace - souvislý sled bloku, informace uložení se skládá pouze z čísla prvniho bloku.
+: lepsi prace s diskem
–: hledani volneho mista, zvetsovani velikosti
Spojovaná alokace – FAT
Indexová alokace – inode – unix

129. Uveďte, jakými způsoby lze na disku ukládat strukturu adresářů. Jednotlivé způsoby ilustrujte na existujících systémech souborů a zhodnoťte.

Triviální případ jednoúrovňového adresáře, koncept ROOTu v MS-DOSu. Hierarchické adresáře, ukládání podadresářů do nadřazených adresářů. DOSácká klasika, totéž u UNIXu. Záznamy pevné velikosti
FAT: Spojový seznam:
–: delší jména (ve FAT32 jako „skryté“ soubory, pokračování jména…)
–: pomalé hledání, Lineární prohledávání (tzn záleží na počtu souborů v adresáři).
B-stromy (Win NT)
+: rychlé hledání

130. Vysvětlete rozdíl mezi hard linkem a symbolic linkem. Porovnejte výhody a nevýhody obou typů linků.

hard link = Odkaz na fyzická data.
symbolic link = Soubor obsahující jméno souboru na které odkazuje.

Hard link nejde vytvářet mezi FS, soft linku to nevadí. Používání hard linků zvýší režii při mazání.

131. Uveďte, jakými způsoby lze na disku ukládat informaci o volných blocích. Jednotlivé způsoby ilustrujte na existujících systémech souborů a zhodnoťte.

Evidence volného místa, seznam volných bloků a bitmapy. Funkce bitmap je jasná, seznam volných bloků se zdá být nevýhodný, až na schopnost mít v paměti jen malou část tablice a přesto uspokojovat velký počet dotazů na volné bloky, a možnost ukládání právě ve volném místě.
Bitmapa – HPFS, EXT
- bloky pevné velikosti
Spojový seznam volných bloků – FAT (respektive ve FAT tabulce jsou to záznamy s 0).
– seznam volných bloků
– spojování uvolněného bloku se sousedy

132. Popište způsob uložení informace o umístění dat souborů v systému souborů FAT. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.

Boot sektor s rozměry filesystému, za ním dvakrát FAT, za ní *ROOT* directory,za ním data area. *Directory entry* obsahuje název, typ (adresář, soubor) a číslo prvního bloku, kde se nachazeji data.
FAT tabulka obsahuje spojový seznam použitých clusterů. Položka FAT tabulky může obsahovat 0 = volný, číslo dalšího clusteru, nebo nějaké jiné značící vadný cluster, konec souboru, …
–: Složitost operací: Chybí indexy adresářů, takže se prochází lineárně a metadata jsou FYZICKY daleko od dat. Fragmentace (vzhledem ke clusterům).
+: Snadná implementace.

133. Popište způsob uložení informace o struktuře adresářů v systému souborů FAT. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.

132); Adresář je soubor, který má nastaven bit; Jako data obsahuje *directory entries*, což jsou informace o souborech, které sám má. Každé directory entry obsahuje název, typ (adresář, soubor) a číslo prvního bloku, kde se nachazeji data.

134. Popište způsob uložení informace o umístění volných bloků v systému souborů FAT. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.

Na začátku jména 0x00=> prázdné, 0xE5=> smazané.

135. Popište způsob uložení informace o umístění dat souborů v systému souborů EXT2. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.

Boot sector, poté je disk rozdělen do groups; V každé groups jsou: FBB (bitmapa volného místa do dat):FIB (bitmapa volného místa), IN (tabulka Inodů): data. Soubory reprezentují INody, pro každý soubor existuje právě jeden INode. Na začátku inodu se nachází různé atributy jako typ, práva, vlastník, časové razítko… Za nimi se nachází odkazy do datové části. Odkazy až na poslední tři jsou přímo na data; odkaz (n-2). je odkaz na tabulku odkazů, (n-1). je odkaz na tabulku, kde je kazdy odkaz odkaz na tabulku a poslední je tříúrovňový. Mapa vadných bloků je udržována v inodu číslo 11.

```
struct ext2_inode {
__u16 i_mode; /* File mode */
__u16 i_uid; /* Owner ID */
__u32 i_size; /* Size in bytes */
__u32 i_atime; /* Access time */
__u32 i_ctime; /* Creation time */
__u32 i_mtime; /* Modification time */
__u32 i_dtime; /* Deletion Time */
__u16 i_gid; /* Group ID */
__u16 i_links_count; /* Links count */
__u32 i_blocks; /* Blocks count */
__u32 i_flags; /* File flags */
__u32 i_block [EXT2_N_BLOCKS]; /* Ptrs to blocks */
__u32 i_version; /* File version for NFS */
__u32 i_file_acl; /* File ACL */
__u32 i_dir_acl; /* Directory ACL */
__u32 i_faddr; /* Fragment address */
__u8 l_i_frag; /* Fragment number */
__u8 l_i_fsize; /* Fragment size */
};
```

136. Popište způsob uložení informace o struktuře adresářů v systému souborů EXT2. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.

Directories are stored either unsorted, or with hash tree indices. Directory entry jako data v Inode.

```
struct ext2_dir_entry_2 {
__u32 inode; /* Inode number */
__u16 rec_len; /* Directory entry length */
__u8 name_len; /* Name length */
__u8 file_type; /* File type */
char name [EXT2_NAME_LEN]; /* File name */
};
```

137. Popište způsob uložení informace o umístění volných bloků v systému souborů EXT2. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.

Bitmapa volného místa per group. Je to blíže k datům, ale zase to není „spojitý“…

138. Popište způsob uložení informace o umístění dat souborů v systému souborů NTFS. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.

Na disku neleží nic než soubory, informace o nich jsou uloženy v MFT aneb Master File Table. Každý soubor je jednoznačně identifikován indexem do MFT, MFT sama je také soubor s indexem 0, další význačné indexy jsou 1 pro MFT mirror, 2 pro transaction log, 3 pro root directory, 4 pro allocation bitmap, 5 pro bootstrap, 6 for bad cluster file atd.
Na začátku disku je jen bootsektor s rozměry disku a pointerem na MFT, jeho kopie je uložena ještě kdesi na (nejspíš konci) partition. Celý zbytek disku se adresuje po clusterech, které jsou podobně jako u FAT mocninné násobky sektoru.
Každý soubor je set of attributes, jeden z atributů je default, což je data stream. Každý záznam v MFT obsahuje magic, reference count, flags, potenciálně pointer na base file record pokud toto je extension file record (když se vše nevejde do base file recordu). Následují: file attributes, u nich záznam obsahuje jméno, typ a data, data mohou být bud resident (data v MFT) nebo non-resident, v tom případě následuje v záznamu run list, což je sekvence bloků clusterů. Adresáre jsou uložené jako soubory, jejichž obsah je B strom s referencemi na obsažené soubory. Mírně zjednodušeno se největší nevýhodou se zdá být fragmentace.

139. Popište způsob uložení informace o struktuře adresářů v systému souborů NTFS. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.

Adresáře jsou uložené jako soubory, jejichž obsah je B strom s referencemi na obsažené soubory.

140. Popište způsob uložení informace o umístění dat souborů v systému souborů na CD. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.

Disk rozdělen na sektory zpravidla 2048 bytes, prvních 16 sektorů prázdných pro bootstrap loader. Zbytek disku je popsán sekvencí volume descriptors; jeden per sektor. Nejdůležitější je Primary Volume Descriptor s adresou root directory, path table a dalšími zbytečnostmi (copyright, abstract, bibinfo). Adresáře jsou usual stuff, name of 30 chars max, attributes, soubor je udán počátečním sektorem a délkou (teoreticky je možné uvést víc adresářových položek pro soubor z více fragmentů, ale nepoužívá se, stejně jako se zdá se nepoužívá interleaving – což je míchání více souborů do jednoho?).
Podpora multisession disků: Deskriptor obsahuje navíc políčko verze, ten s nejvyšším číslem verze vyhrává. Deskriptory můžu ukládat na víc míst (například na konec).

141. Popište způsob uložení informace o struktuře adresářů v systému souborů na CD. Uveďte přednosti a nedostatky tohoto způsobu uložení informace.

Aby se adresáře nemusely prohledávat item by item, uloží se do path table seřazený (podle hloubky a dalších kritérií) seznam všech cest na disku, pro každou cestu obsahuje path table sektor příslušného adresáře a jeho parenta.

142. Vysvětlete princip integrace více systémů souborů v operačním systému do jednoho prostoru jmen.

Virtual File System (rozhraní, reálný OS implementuje toto rozhraní). => podpora více file systémů pod jedním Operačním Systémem.
Ve Windows může mít každý logický disk svůj FS.
Provedení v Linuxu je přímočaré. Při volání open se systém podívá na začátek jména souboru a podle toho zda je absolutní či relativní vezme dentry bud' root directory nebo current directory. Pak už se jen postupně parsuje jméno a každá jeho část se zkusí najít v dentry cache, pokud tam není, tak se použije lookup funkce parent dentry.
Do tohoto mechanizmu celkem přímočaře zapadá i mounting. Pokud se do adresáře něco namountuje, jeho dentry bude obsahovat pointer na root dentry namontovaného file systému. Tento dentry zůstane díky busy locku (caused by bus locking and cache invalidation when test-and-set operation needs to access memory atomically) vždy v dentry cache. Při parsování cesty se pak u každého dentry ještě kontroluje, zda nemá mounted file systém, pokud ano, vezme se jeho root dentry. Navíc mám negativní dentry – „tenhle soubor neexistuje".

Stackable FS (Union FS)
Obsahuje disjunktní podstromy (pro každý FS), celkem složí celý strom. Vrstvový model.

143. Popište socket jako abstrakci rozhraní operačního systému pro přístup k síti podle Berkeley sockets. Uveďte základní funkce tohoto rozhraní včetně hlavních argumentů a sémantiky.

The Berkeley socket interface centers around the concept of a socket as an object that facilitates communication. The socket can be bound to a local address and connected to a remote address. Data can be sent and received over a socket.

```
int socket (int domain, int type, int protocol);
```
Domain specifies socket protocol class.
Type specifies socket semantics.
Protocol specifies socket protokol.
The socket call creates the socket object. An error is returned if the combination of class, type, protocol is not supported.

```
int bind (int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```
The bind call binds the socket to a given local address. The binding is typically necessary to tell the socket what local address to listen on for incoming connections.

```
int listen (int sockfd, int backlog);
```
The listen call tells the socket to listen for incoming connections and sets the length of the incoming connection queue.

```
int accept (int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```
The accept call accepts an incoming connection on a listening socket that is SOCK_STREAM, SOCK_RDM, SOCK_SEQPACKET. The function returns a new socket and an address that the new socket is connected to and keeps the original socket untouched.

```
int connect (int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```
The connect call connects a socket that is SOCK_STREAM, SOCK_RDM, SOCK_SEQPACKET to a remote address, and sets a remote address of the socket otherwise.

```
ssize_t send (int sockfd, const void *buf, size_t len, int flags);
ssize_t sendto (int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);
```
The send family of calls sends data over a socket. Either the socket is connected or the remote address is specified. The write call can also be used but the flags cannot be specified in that case.

```
ssize_t recv (int sockfd, void *buf, size_t len, int flags);
ssize_t recvfrom (int sockfd, void *buf, size_t len, int flags,
struct sockaddr *from, socklen_t *fromlen);
```
The recv family of calls receives data over a socket. The read call can also be used but the flags cannot be specified in that case.

144. Vysvětlete účel funkce select v rozhraní operačního systému pro přístup k síti podle Berkeley sockets.

```
int select (int setsize, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
int poll (struct pollfd *ufds, unsigned int nfds, int timeout);
```
The select call is used to wait for data on several sockets at the same time. The arguments are sets of file descriptors, usually implemented as bitmaps. The file descriptors in readfds are waited for until a read would not block, the file descriptors in writefds are waited for until a write would not block, the file descriptors in exceptfds are waited for until an exceptional condition occurs. The call returns the number of file descriptors that meet the condition of the wait. The poll call makes it possible to more precisely distinguish what events to wait for.

145. Vysvětlete, k čemu slouží sockety v doméně PF_UNIX .

PF_UNIX - local communication

146. Vysvětlete, k čemu slouží sockety v doméně PF_NETLINK .

PF_NETLINK - kernel communication

147. Popište princip funkce mechanismu vzdáleného volání procedur a načrtněte obvyklou architekturu jeho implementace.

RPC) is a technology that allows a computer program to cause a subroutine or procedure to exekute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding (STUBS) the details for this remote interaction.
RPC is an easy and popular paradigm for implementing the client-server model of distributed computing. An RPC is initiated by the client sending a request message to a known remote server in order to execute a specified procedure using supplied parameters. A response is returned to the client where the application continues along with its process.

148. Vysvětlete, proč při implementaci přístupu k síti v operačním systému může kopírování přenášených dat být problém. Zhodnotťe míru tohoto problému a uvedťe, jakým způsobem jej lze odstranit.

The individual modules that process packets may need to add headers or footers to the data, which may prompt a need for moving the data to make room for the headers or footers. With top desktop systems moving data in memory in hundreds to thousands of MB per second and top network systems moving data in wires in thousands of MB per second, even a small amount of data copying may be a problem.
The second pitfall is caused by excessive data dispatching. Many solutions exist, the traditional ones including hash tables, the wilder ones ranging from dispatcher shortcut caching to dispatcher code generation and dispatcher code upload.
Both pitfalls can be sidestepped by using smart hardware.

149. Vysvětlete roli filtrování paketů v operačním systému. Uvedťe příklady kritérií, podle kterých mohou být pakety filtrovány a příklady akcí, které mohou filtry s pakety vykonávat.

The networking layer must decide what to do with each packet. A packet can be delivered to a local recipient, forwarded to a remote recipient, or even dropped. This mechanism is configurable to avoid abuse of default rules for delivering, forwarding, discarding.
Each point contains a sequence of rules. A rule can classify packets using information from packet header (source and destination address, protocol) or from packet processing (source and destination interface). Modules that classify packets can be added, available modules include file conditions, connection marks, connection rates, connection state, security context, random and others.
Actions:
ACCEPT means process packet, DROP means discard, QUEUE means queue for user space application to decide, RETURN means continue previous chain.
Filtrovani vstupu.

150. Vysvětlete roli plánovače paketů v operačním systému.

Časuje output packety (předcházení přeplněni front a sítě).

151. Popište Token Bucket algoritmus pro plánování paketů a vysvětlete, co je jeho cílem.

Když mám jeden tok na obsloužení pásma. Tok je jako bucket pevné velikost. Data jsou pravidelně přidávány (od procesů) odebírány po odeslání. Když nějaká přijdou a bucket je plný, nic se nepřidá. Nic se nepošle, když je bucket prázdný. Rychlost přidávání závisi na šířce pásma a kapacita bucketu na limitu proudu (kyblík (bucket), do kterého kape voda (tokeny)).

152. Popište Stochastic Fair Queuing algoritmus pro plánování paketů a vysvětlete, co je jeho cílem.

SFQ se používá v situaci, kdy se spousta datových toků pere o jednu šířku pásma. Funguje to tak, že algoritmus se navenek (pro datové toky tváří), že každý má svou frontu, do které cpe data, a ty se RR procházejí a odesílají se z nich data. Ve skutečnosti má ale relativně málo front a identifikace jednotlivých toků na ně hašuje. Protože takhle samozřejmě může docházet ke kolizím, tak čas od času mění hašovací funkci, aby žádné dva toky příliš dlouho nepadaly do stejné fronty.

153. Popište Class Based Queuing algoritmus pro plánování paketů a vysvětlete, co je jeho cílem.

Vytvořím si strom pro celou síť, jehož uzly budou typy provozu (kazza, web server, …) To větvím do kolika úrovní chci. Každý uzel má stanovenu úroveň, kolik z celkové kapacity sítě může využít. Vždy platí, že součet uzlů v podstromu dá úroveň kořene. Každý uzel může aktuálně využívat síť (méně, akorát, více); Zdroj dat, což jsou listy, mohou být (ne)spokojeny. Úrovně rostou směrem od listů => kořen má nejvyšší lvl.
Zpráva se odešle, pokud:

- Využívám síť méně, nebo
- Existuje předek, který využívá síť méně na úrovni I & !existuje nespokojený uzel na úrovni <=I.

154. Popište Random Early Detection algoritmus pro plánování paketů a vysvětlete, co je jeho cílem.

Spíše pro routery; mechanismus předcházení zahlcení sítě.
Chce se vyhnout vyhazování paketů z konce front, kdyz uz je plna. Udržuje se minimální, maximální a průměrná délka fronty. Pakety jsou označeny nebo vyhozeny s pravděpodobností aktuální pruměrné délky fronty. Dává to brzké varovaní naplněni fronty.

155. Popište architekturu síťového systému souborů NFS včetně používaných protokol ů a hlavních operací těchto protokolů.

Mount a NFS protokol, oboje postaveno nad RPC.
Mount protokol dovoluje klientovi poslat mount request na server. Server odpoví zasláním file handle na mounted directory (operace jsou MNT, UMNT, UMNTALL, DUMP mount list, EXPORT export list). File handle by měl čistě teoreticky být opaque 32 bytes pro klienta. Typicky obsahuje: file system ID, I-node, generation ID. NFS protokol pak nabízí běžné souborové operace s výjimkou open a close, protože je stateless (GET/SET na atributy, LOOKUP, READ, WRITE, CREATE, REMOVE, MK/RM na directories ...)
Verze 3 již obsahuje zámky (s timeouty), stále bezstavová.
Verze 4 opouští bezstavovost, podobné jako SMB. Autentifikace, lepší cachování, agregace operací (jako předchozí), zámky (které může systém zrušit, kdykoliv se mu zlíbí).

156. Vysvětlete, co to je a jaké položky zpravidla obsahuje file handle v síťovém systému souborů NFS.

File system ID, I-node, generation ID (změny souboru).

157. Vysvětlete, jaké problémy přináší bezstavovost NFS při testování přístupových práv a jak jsou tyto problémy řešeny.

UNIX je standardně testuje pouze při otevření, NFS musí pořád (jako řešení se permissions při otevření testují na klientovi a sdílí se prostor UID a GID a relaxují se některé kontroly (vlastník souboru může vše, právo execute implikuje právo read)).

158. Vysvětlete, jaké problémy přináší bezstavovost NFS při mazání otevřených souborů a jak jsou tyto problémy řešeny.

Další je mazání otevřených souborů (opět se řeší na klientovi). Může se mi stát, že mi někdo pod rukama smaže můj otevřený soubor. Může se vyřešit například zámky, ale pouze od V3.

159. Vysvětlete, jaké problémy přináší bezstavovost NFS při zamykání souborů a jak jsou tyto problémy řešeny.

Problém může nastat, když „to“ spadne. Proto se zámky přidělují pouze na omezenou dobu (max na LEASE). Po restartu je systém ve speciálním módu, kdy pouze potvrzuje prodloužení zámků (a tak si je rekonstruuje), avšak odmítá zámky nové. V tomto stavu je po dobu LEASE. Poté pokračuji, jako kdyby se nic nestalo. Samozřejmě, pokud někdo bude chtít škodit (obnovit zámky na vše), tak může!

160. Vysvětlete termín authentication .

Authentication je problém ověření toho, zda je aktivita (proces, uživatel) tím, za koho se vydává. Zpravidla se používá kombinace jména a hesla, typicky je pak v systému nějaká centrální autorita, která toto ověřuje, ostatní se už jen ptají téhle autority.

161. Vysvětlete termín authorization .

Authorization je problém rozhodnutí, zda je daná aktivita (proces, uživatel) oprávněna udělat nějakou akci nad nějakým prostředkem (soubor, zařízení).

162. Vysvětlete, co to je access control list .

Access control lists je technika, kde se s každým prostředkem uloží seznam aktivit a jim dovolených akcí. ACL umí leckteré UNIXy, v těch jsou zpravidla jako aktivity bráni users nebo groups a akce jsou klasické RWX nad soubory. Z téhož principu vlastně vycházejí i standardní atributy u UNIX souborů.

163. Vysvětlete, co to je capability .

Capabilities je technika, kdy si každá aktivita (user) nese seznam prostředků a nad nimi povolených akcí. Při přístupu k prostředku se pak aktivita prokáže svou capability, kterou systém verifikuje. Toto je mechanizmus, který běžné systémy příliš často nemívají, ale u distribuovaných systémů nachází značné uplatnění, příklady jsou capabilities u Amoeby, Machu či EROSu nebo credentials v CORBE.