Thema 1: OS Übersicht
Thema 2: Concurrency in Threads
Thema 3: Kommunikation zwischen Prozessen
Thema 4: Linux, Windows, etc.
Thema 5: Deadlocks

Thema 1, OS Übersicht: User-Prozesse, OS/Kernel Prozesse (System Calls), Hardware (CPU Cores, Memory, Storage, other I/O, Driver), Prozesse (Memory, Ressourcen, File Handles) und Threads (PC, Register, Ausnahme: Threadlocal), Wie startet man Prozesse (Shell, Doppelklick, fork(), exec(), Parent, Tree) und Threads (new Thread). Wie tötet man Prozesse (kill, aus beliebigen Anwendungen raus). Verwaltung mittels PCB Process Control Block, Queues von wartenden Prozessen, Thread Scheduling Basics, Non-Preemptive/Preemptive

Q: Could you explain the difference between threads, process and CPUs
A: Processes have their own address space, which they exclusively manage (Virtual Memory). They may use shared memory to map part of another process's memory into their own memory, but that is selective. Threads of a process share that process' memory, and simply provide a way to perform concurrent operations inside a single process. CPUs are the physical hardware that executes the program code of a process.

Q: Hat Unix auch die Unterscheidung zwischen Prozessen und Threads? Ich habe mal irgendwo gelesen, dass unter Unix alles gleich behandelt wird. Oder gilt das nur für das Scheduling?
A: Ein Prozess ist im Prinzip ein Programm, das gestartet wurde. Jeder Prozess hat seinen eigenen Speicherbereich im Arbeitsspeicher und kann nicht auf den Speicher eines anderen Prozesses zugreifen. (Ausser mit root-Rechten.) Ein Prozess enthält mehrere Threads, die den Programmcode ausführen. Alle Threads eines Prozesses teilen sich den Speicherbereich des Prozesses. Auf dem Prozessor werden Threads ausgeführt, da Prozesse nicht direkt ausgeführt werden können. Deshalb verwaltet der Scheduler Threads. Java direkt Systemthreads, Python nur ein Thread, scheduling intern.

Q: Was ist der **PCB** genau und was ist seine tiefere Aufgabe?
A: Verwaltung der Prozesse durch OS.

Q: (2x) Was genau ist die Bedeutung von "**preempted**"?
A: Preempted = Unterbrochen. In allen modernen Betriebssystemen (OS) können Prozesse/Threads unterbrochen werden, z.B. wenn ein Interrupt behandelt werden muss. Aeltere OS (z.B. Apple's OS 9) waren non-preemtive: wenn ein Prozess die Kontrolle über den Prozessor hatte, dann konnte nichts und niemanden diesen aufhalten. Sogar das OS selbst (Kernel) ist heute preemptive.

Q: Shared memory requires that two or more processes agree to remove the restriction of not accessing another process' memory. What kind of vulnerability can this lead to in a system?
A: How is shared memory access implemented? Access for designated other processes, or access for all processes that know a particular "key"? As all processes of that user can access the shared memory, the processes should naturally not store any sensitive data there, which may not be known to all processes of that user. Further, processes reading from shared memory need to trust other processes of that user not to wreck the content.

Q: (2x) Was ist der Unterschied zwischen Preemptive-Kernels und Non-preemptive Kernels? (warum keine Race condition im Preemptive?)
A: Preemptive bedeutet, dass Prozesse vom Kernel während der Ausführung unterbrochen werden können, ihre Ausführung also vom Kernel verwoben werden kann. Non-preemptive bedeutet, dass jeder Prozess, der einmal gestartet wurde, selber "entscheidet", wann er aufhört oder unterbricht. Wenn ein User-Prozess einen unendlichen Loop hat, kann man den Rechner abstellen?

Q: In Abschnitt 3.6.3 Pipes in Figure 3.24 ist der Programmcode von Ordinary Pipes gezeigt. Der Sinn hinter Pipes ist es, soweit ich es verstanden habe, dass das ursprüungliche Programm mit dem Childprogramm kommunizieren kann (hier: Vater schreibt, Kind lies). Allerdings ist im Code:
if (pid>0) { /*parent process*/...}
else {/*child process*/}
Mit if und else geschrieben. D.h. entweder schreibt das Vaterprogramm oder das Kindprogramm liest. Wie kann jetzt aber das Kind lesen, wenn der Vater geschrieben hat?
A: In C (Unix/Linux) kann sich ein Prozess mit dem System-Call **fork()** in zwei Prozesse aufteilen, beide haben die gleichen Variablen und den gleichen Code. Wie kann man nun erreichen, dass nicht beide Prozesse genau das gleiche machen? Man kann den Return-Value von fork() prüfen. Dieser gibt die Prozess ID des eigenen Kindes zurück, d.h. Ist der Wert > 0 ist man der Parent, ist der Wert = 0 ist man das Kind selbst (man hat noch keine Kinder)

Q: Seite 12, "if (pid > 0) [...]": was passiert beim **Fork**ing? Welche Werte nimmt pid bei Parent und bei Child an? Das else-statement würde ja nur ausgeführt, wenn pid==0?
A: Fork retourniert zwei verschiedene Werte je nachdem man im Child ist oder im Parent (fork wird einmal im Parent aufgerufen, returned aber zweimal, einmal im Child und einmal im Parent). Der Parent bekommt die Prozess ID des Childs und das Child bekommt 0, damit koennen wir die beiden Prozesse unterscheiden und der Parent kennt auch gleich das Child.

Q: Seite 11, Mitte: **fork():** was macht dieser Syscall genau? Erstellt es eine exaktes Abbild des Prozesses mit demselben Status? Oder fängt das Child wieder beim Startpunkt (Anfangs main()) an? Je nachdem ergibt die Aussage, dass offene Dateien vererbt werden, mehr oder weniger Sinn...
A: Fork erstellt ein exaktes Abbild des Prozesses in dem es aufgerufen wird, das beinhaltet Memory pages, File descriptoren, und Prozess Counter. Heisst also dass sowohl der neue als

auch der alte Prozess genau an dieser stelle weiterfahren. Danach kann dann ein beliebiges Programm nachgeladen werden und die geteilten Resourcen selektiv geschlossen werden.

Q: Auf Seite 23 im PDF steht «The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified.» Wie ist das gemeint? Angenommen ein I/O-Device löst einen Hardware-Interrupt aus, dann können wir den doch nicht einfach ignorieren (wir würden ja den Event komplett verpassen)?
A: meistens ist es so dass die Interrupt handler nur das Event im Kernel registrieren damit es spaeter abgearbeitet werden kann und wir dieses nicht direkt verarbeiten.

Q: How does Scheduling Work?
A: Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
Rule 2: If Priority(A) = Priority(B), A & B run in Round Robin.
Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).
Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced.
Rule 5: After some time period S, move all the jobs in the system to the topmost queue.

Q: How can we give **priorities to our processes**? How do I see them on my OS (root, user)?
A: Priorities are a simple numeric value attached to the process information tracked by the kernel. It is used by the kernel as a weight when selecting the next process to schedule, so that processes with a higher priority are scheduled more often than lower priority processes. The priority is also called niceness in Linux/Unix and can be inspected using the ps or top tools. Some Kernel processes have a high priority assigned by the OS, for user processes a user can often decide the priority herself. Linux "nice", Windows Task Manager.

Sonderthema 1b, Priority Inversion

Q: Warum wird bei **Priority Inversion** nicht der höhere Prozess zuerst drangelassen? Gibt es bei Locks nicht Varianten die die Priorität von Prozessen berücksichtigen, oder wenn der Prozess unterbrochen wird, warum kann der wichtigere Prozess ihn nicht unterbrechen? Was ist wenn nun ein noch wichtiger Prozess kommt?
A: Ich bin nicht sicher, ob ich die Frage verstehe. Vielleicht hilft eine einfache Einführung über Scheduling? Natürlich unterbricht ein wichtiger Prozess einen unwichtigen, aber das Problem bei Priority Inversion ist, dass das wichtige durch einen noch unwichtigeren Aufgehalten wird, i.e. 2 (läuft) < 3 (wartet auf 2) < 1 (wartet auf 1).

Q: **Priority Inversion**: Kurze Skizze der Reihenfolge und Priority Levels? Wenn L die Ressource R freigibt, wird die Priority von L wieder zurueckgesetzt. Aber ist L dann nicht meist eh schon fertig? [diese Frage/Antwort am besten mit dem vollen Beispiel kombinieren]
A: Dann ist L mit der Ressource fertig und ein anderer Thread (z.B. H) kann sie verwenden. Genau dann und deswegen benötigt L die erhöhte Priorität nicht mehr.

Q: Ich wäre froh um ein Beispiel zur **priority inversion**, ich kann mir das nicht so recht vorstellen…
A: Man stelle sich der Einfachheit halber ein Einprozessorsystem vor. Es laufen 3 Threads mit den Prioritäten 1 (niedrigste), 2 und 3 (höchste). Der Thread mit Priorität 1 erhält ein Lock (für einen kritischen Abschnitt oder eine Ressource) und arbeitet damit vor sich hin. Nun kommt der Thread mit Priorität 3 und will das Lock ebenfalls haben, muss jetzt aber warten. Das Betriebssystem markiert sich den Thread 3 als nicht schedulebar bis das Lock verfügbar ist. Wenn das Betriebssystem nun den nächsten Thread zum Ausführen wählen muss, hat es also nur die Threads mit den Prioritäten 1 und 2 zur Wahl. Folglich wird der Thread mit Priorität 2 mehr ausgeführt als der mit Priorität 1, auf den Thread 3 wartet. Die Prioritäten von Threads 2 und 3 wurden also quasi “invertiert”. Lösung: Priority-Inheritance.

Q: (2x) Der Unterschied zwischen Spinlocks/Mutexes und Semaphores ist einigermassen einleuchtend, was ist aber eine condition variable? Können sie die verschiedenen Synchronisationsmechanismen auf Benutzerebene nochmal kontrastieren? After all these concurrency mechanisms I lost the overview… What are the differences between a mutex and semaphore? Are they doing exactly the same and is there only a difference in the implementation? (Frage leitet Thema 2 ein.)

Thema 2, Concurrency in Threads: Zusammenfassung wie man Concurrency in Threads organisiert: Ganz unten sind die Hardware-Primitive (Test-and-Set, Swap, LL/SC, auf-condition-warten, etc.). Damit implementiert man Locks für Mutex (Spin-Locks und Queue-Locks, Kapitel 4 der Vorlesung), oder auch die höherwertigen Semaphoren, Java-Synchronized, Pipes. Und schliesslich löst man damit dann die Probleme, die so auftreten (Producer-Consumer, Writer-Multireader, Probleme wie Dining Philosophers).

Q: Message-Passing Systems: What protocol is used to establish the communication link?
A: Depends on OS, examples are shared memory, sockets or pipes.

Q: Why does the Peterson’s Solution not work on modern computer architectures?
A: Reordering of instructions. Probably the keyword volatile does help, though.

Q: (2x) Mutex: Mir ist die Bedingung "Progress" im Abschnitt 6.2 nicht ganz klar.
A: Der wichtigste Teil dieser Bedingung ist, dass, wenn Prozesse einen leeren kritischen Abschnitt betreten wollen, sie das früher oder später tun können. Der zweite Teil der Bedingung ist, dass das nötige Kooperieren für das Betreten nicht von unbeteiligten Prozessen abhängen sollte.

Q: S. 42, oben: wenn ein Prozess (in einem Singleprozessorsystem hier) nicht unterbrochen werden darf, solange er ein Lock besitzt, wie kann ich als OS den Counter nachführen, wenn der Prozess ein weiteres Lock requestet?
A: Die System calls preempt_disable und preempt_enable setzen lediglich das interesse dass man bei locks preempted werden soll oder nicht, der preempt_count deaktiviert dann tatsaechlich die preemption. Ein lock zaehlt atomar hoch oder runter wenn er ein lock acquired oder released, mit hilfe einer RMW operation. Wenn der kernel preempted wuerde dann kontrolliert er erst den preempt_count und unterlaesst das preempten wenn dieser nicht 0 ist.

Q: How does a process decide to whether it should "**sleep or busy-wait**"? What about the sleep() system call?
A: In most cases, sleeping is the preferred answer: sleeping incurs some overhead through the context switches and possibly some delay in re-scheduling, but busy-waiting potentially wastes far more CPU cycles. In special cases, where minimizing the time to the acquisition of the lock is paramount, busy-waiting may be acceptable. There are also various hybrid approaches: for example, first trying to busy-wait for a bit but after some time sleeping instead, or the “adaptive mutex” approach offered by Solaris: busy-waiting if and only if the lock holder is known to currently be running on another core. Sleep puts the process to sleep until the timer fires.

Q: S. 42, oben, letzter Abschnitt zu Linux: Terminologie: Spinlocks sind doch eine mögliche Implementierung einer Semaphore, gemäss S. 28, Abschnitt 6.5.2 unten? Wie unterscheiden sich in 6.8.3 nun also die Spinlocks von Semaphores?
A: Spinlocks sind eine unterklasse von Semaphoren. In modernen Betriebssystemen signalisiert eine Semaphore dass der Thread/Prozess nicht gescheduled werden kann solange die Semaphore nicht frei ist. Spinlocks hingegen verbrauchen rechenzyklen zum spinnen auch wenn sie nicht weiterfahren koennen.

Q: Wie sind die Semaphorefunktionen genau implementiert? Sind sie jetzt in der Hardware drin oder nicht?
A: Die Semaphore ist ein Softwareobjekt; weil aber wait() und signal() atomar ausgeführt werden müssen, benötigt man für die Implementierung atomare Operationen wie test-and-set. In manchen RISC-Architekturen werden auch schwächere atomare Operationen in Hardware implementiert, die nur erkennen können, wenn mehrere Prozesse concurrently wait() oder signal() aufrufen.

Q: Ende von 6.5.2 (Semaphore Implementation): Why is busy waiting moved to the critical sections of the application programs? I thought the other processes are then blocked and no busy waiting occurs?
A: We have to ensure that the wait and signal calls used for the Semaphore are atomic, i.e., no two processes may call signal or wait at the same time, hence we need to ensure mutual exclusion when performing these calls.

Q: Es wird gesagt, dass die Hardwarefunktionen zu kompliziert seien. Werden sie nun gar

nicht benutzt? (Section 6.5: “The hardware-based solutions to the critical section problem presented in Section 6.4 are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a semaphore.”)
A: Ähnlich wie vorherige Frage. Die Schwierigkeit von TestAndSet, CompareAndSwap, etc. ist, sie richtig zu benutzen, d.h. zu garantieren, dass das Programm unter allen Bedingungen korrekt ist. (-> ABA-Problem zeigen?) Die Primitive sind häufig in (Software-)Libraries verpackt, die Funktionen wie Semaphoren oder Locks zur Verfügung stellen. Meistens ist es ausreichend und einfacher, diese Funktionen zu verwenden. Aber wenn man z.B. selbst eine concurrent list programmieren will, dann möchte man nicht die ganze Liste locken, wenn man etwas ändert, sondern nur einen Pointer umhängen. Dann ist es viel effizienter CAS zu verwenden. Für lower level concurrency werden also TAS und CAS bestimmt in der Praxis verwendet.

Q: Wo „ist“ das Lock? Ist es teil des Programm oder ein globales Objekt. Gleiches gilt auch beim Semaphore.
A: Eine Lockingstrategie ist ein Mechanismus, der mutual exclusion zur Verfügung stellen soll. Den Mechanismus kann man auf verschiedene Weisen implementieren - als Objekt, das von Prozessen geteilt wird, also Zugriffsmethoden auf eine synchronisierte Datenstruktur, etc. Ein Lock selbst ist kein konkretes Softwareobjekt, sondern die Eigenschaft, dass ein Prozess Zugriff auf die gelockte Ressource hat. Wenn die Lockingstrategie korrekt ist und richtig implementiert wurde, dann garantieren Locks, dass das critical section Problem auch tatsächlich gelöst wird. Man spricht davon, dass ein Prozess ein Lock hält, eines anfragt, eines wieder hergibt (release) etc., was alles Beschreibungen dafür sind, in welchem Schritt des Eintritts in eine critical section sich ein Prozess gerade befindet. Eine Semaphore ist ein Objekt, das einen counter für “Anzahl Prozesse, die noch gleichzeitig in ihre kritische Sektion eintreten dürfen” verwaltet. Der counter wird in die Semaphore gewrapped, damit potentiell nebenläufige Inkrement-/Dekrementoperationen darauf synchronisiert werden können. Eine Semaphore, die für eine Menge von Prozessen sicherstellt, dass nur höchstens eine bestimmte Anzahl von ihnen gleichzeitig in ihrer critical section sind, wird von den Prozessoren in der Menge geteilt. Die Semaphore sucht nicht aktiv nach Prozessen, die verwaltet werden müssen; es ist Aufgabe des Programmierers sicherzustellen, dass alle Prozesse die Semaphoren nutzen, die einander in die Quere kommen könnten.

Q: (2x) Ich würde gerne noch einmal erklärt haben, wie ein **Readers-Writer-Lock** funktioniert.
A: Im Unterschied zu einem Lock, das immer nur von einem Prozess gehalten werden kann, kann ein RW-Lock von beliebig vielen Readers gleichzeitig gehalten werden. Ein RW-Lock ist gut, wenn viele Reads aber wenige Writes gemacht werden. Da die Reader keine Daten verändern, können beliebig viele gleichzeitig lesen. Dagegen, kann jeweils nur ein Writer das Lock kriegen. Read und Write schliessen sich auch gegenseitig aus. Das Lock ist also in einem von zwei Modi: 1) Read: beliebig viele Reader, 2) Write: genau ein Writer. Ein RW-Lock kann entweder Reads oder Writes bevorzugen. Wenn Reads bevorzugt sind, kann es aber sein, dass die Writes verhungern (Implementierung in 6.6.2). Wenn Writes bevorzugt werden, ist weniger concurrency möglich (dafür sind aber die Daten immer aktuell). Priority heisst, dass solange ein bevorzugter Prozess Zugriff möchte, Prozesse vom anderen Typ das Lock nicht erhalten.

Q: Zu der letzten Vorlesung: Dort wurde gesagt, dass TAS-Locks ineffizienter sind als TTAS-Locks, da der TAS-Befehl einen Schreibbefehl beinhaltet während er spinnt, wohingegen der TTAS-Lock ja keinen Schreibbefehl ausführt bei Spinnen. Allerdings verstehe ich dieses Argument nicht ganz, da ja auch bei der TAS-Anweisung der Schreibbefehl nur dann ausgeführt wird, wenn das Lock tatsächlich an sich genommen werden kann, oder verstehe ich da etwas falsch? Dann würde ja TAS genau das selbe machen wie TTAS (von der Anzahl Schreibzugriffe her)…
A: TAS wird als schreibbefehl gewertet auch wenn er den Wert nicht tatsaechlich aendert. Es waere moeglich dass TAS nicht ueber den Bus geschickt wird wenn der Cache nicht dirty ist, aber das bringt nicht viel: wenn 3 Prozesse (P1, P2 & P3) locken, dann bekommt der erste (P1) das Lock, die anderen haben einen dirty cache. Jetzt fuehrt P2 ein TAS aus, kann sich den Wert cachen und spinnt auf seinem Cache. P3 muss TAS auch ausfuehren, schickt es an den Bus weil sein cache dirty ist, mach damit aber den cache von P2 wieder dirty, und so weiter.

Thema 3, Kommunikation (Concurrency) zwischen Prozessen (sogar verteilt): Pipes & Sockets

Q: S. 15: im Kommandozeilenbeispiel leitet eine Pipe den Output des einen in den Input des anderen Programs um. Die Programme müssen sich aber nicht selbst um die Pipe kümmern. Was macht also der |-Befehl?
A: Die Shell ist der Parent der Prozesse die mit der Pipe verbunden sind, als solches erstellt die Shell die Pipe, forkt um die zwei Prozesse zu kreieren und ersetzt den Stdout FD des ersten Prozess mit dem Input FD der Pipe, und den Stdin FD des zweiten Prozesses mit dem Output FD der Pipe. Somit kann das erste Programm normal auf Stdout/Input der Pipe schreiben und der zweite Prozess liest ganz normal vom Stdin/Output der Pipe.

Q: Warum braucht eine **Pipe** zwei Enden die File Objekte sind? Schreibt man nicht konzeptionell auf ein File und liest dieses wieder?
A: Eine Pipe entspricht schon eher einer einzelnen Datei als zwei, aber Lese- und Schreibe-Handle entsprechen eher zwei Filedescriptors. Z.B. besitzt ein Filedescriptor typischerweise nur einen einzelnen Positionswert für Lesen und Schreiben. Bei Pipes operiert dieser Wert auf dem Pipe-internen Puffer. Wenn die Lese- und Schreibepointer des Puffers immer auf dieselbe Position zeigen müssten, würden Pipes nicht funktionieren.

Q: About **Pipes**: used for communication between different processes, but how and when? Why do they always have these child processes? And the main difference on windows? Can we just pipe character strings or is it possible to pipe predefined data which will be recognized?
A: Pipes are a way to implement interprocess communication, by having one process write to a pipe and the other read from the pipe. The data that can be passed around is in the form of bytestreams, hence anything that can be serialized to a bytestream can be passed through pipes. Unnamed Pipes may only be shared between processes in the same process tree, in which the tree's root created the pipe, e.g., between a child and a parent or between siblings.

This is because the sharing is done using file descriptors. Named pipes are created on the filesystem and can be shared by any process which has access to the filesystem.

Q: Wie geht die Pipe-Eroeffnung Parent-Child in Windows?
A: Der CreateProcess syscall kann angegeben werden ob der Child Prozess auf die FDs (file descriptors) des Parents zugreifen kann. Falls ja werden die Pipe FDs auch vererbt.

Q: Seite 11, oben: Klärungsbedarf: in welchem Fall (oder OS) sind Pipes primär bidirektional? Oder sind sie in jedem Fall unidirektional und eine bidirektionale Pipe wird aus zwei unidirektionalen zusammengestiefelt?
A: Unix Pipes sind unidirectional, Windows pipes sind bidirectional. Unix macht zwei unidirectional Pipes auf um bidirectional semantics zu haben.

Q: Seite 11, Figure 3.22: ich verstehe noch nicht, welche Operationen mit dieser Pipe (und damit fs(0) und fd(1)) erlaubt wären (abgesehen vom Verwendungsbeispiel später), was wäre semantisch/syntaktisch korrekt? Z.B. Wo geschrieben und wo gelesen werden darf und was der Effekt der Operation ist.
A: Schreiboperationen auf FD[0] sind nicht erlaubt, Leseoperationen auf FD[1] sind nicht erlaubt. Der Grund fuer zwei FDs ist dass jeder FD eine eigene Position im File haben kann, wenn wir denselben FD wiederverwenden wuerden, dann wuerde eine Schreiboperation auch die position des lesenden Prozesses beeinflussen. Hinzu kommt dass dadurch dass die FDs semantisch getrennt sind, kann der schreibende Prozess mittels EOF das Ende seiner Arbeit kommunizieren, was ansonsten nicht moeglich waere.

Q: Seite 11, unten (Pipes in Unix): Letzter Satz "Although the program shown in Figure 3.23 [...]" verstehe ich im Zusammenhang mit obiger Frage nicht.
A: Der schreibende Prozess schliesst seinen FD, wodurch der Pipe das Ende der Schreiboperationen signalisiert wird und dadurch kann die Pipe ein EOF zurueckliefern wenn der lesende Prozess das Ende erreicht. Ansonsten waere es nicht moeglich das Ende der Schreiboperationen von “wir warten noch auf neuen Input” zu unterscheiden.

Q: Was ist der Unterschied zwischen Connection-oriented socket und einem Connectionless socket?
A: Verbindungslose Sockets (z.B. UDP) müssen sich nichts weiter merken als die Zieladresse, die für Pakete, die über diesen Socket gesendet werden, verwendet werden soll. Verbindungsorientierte Sockets (z.B. TCP) müssen zusätzlichen Zustand speichern, z.B. die aktuellen Sequenznummern, Daten, die zwar abgeschickt aber noch nicht acknowledged wurden, und Daten, die out-of-order angekommen sind. Das bedeutet auch, dass man verbindungsorientierte Sockets explizit schließen kann (und muss), um eine Verbindung zu beenden, während verbindungslose Sockets das Konzept einer “offenen Verbindung” natürlich nicht kennen.

Q: Wie funktioniert die Kommunikation des Clients und Servers via Loopback?
A: Betriebssysteme implementieren die Loopback-Adressen typischerweise über ein virtuelles Netzwerkinterface namens “Loopback-Adapter” o.ä.. Pakete an die Loopback-Adresse (127.0.0.1 bzw. ::1) werden über dieses Interface “geroutet”. Das Interface macht nichts anderes, als diese ausgehenden Pakete wieder als eingehende Pakete an die Maschine anzumelden. So kann Client-/Server-Software, die für den Netzwerkbetrieb konzipiert wurde, lokal (ohne Codeänderung, transparent) getestet werden.

Q: beim Shootout Fachpraktikum hat sich des öfteren mal die Verbindung meines Computers aufgehängt. Als Resultat habe ich Null vom readline() zurückgekriegt, wie auch im Samplecode der Lektüre korrekt getestet (Seite 10, Figure 3.20, Zeile 13). Wie kommt es dazu?
A: Das ist die Art und Weise wie das Ende des Streams signalisiert wird, wenn man Strings ausliest.

Thema 4, Konkrete Architekturen: Linux, Windows, Sonstiges

Q: Kapitel 6.8.1 (Solaris): Why is [adaptive mutex lock] likely to finish soon? Couldn't it be a process that is running for a long time? e.g. matrix multiplication on large matrices?
A: Adaptive Mutexe sind nicht geeignet für kritische Abschnitte, die mehr als ein paar hundert Instruktionen lang sind; dann wird das Busy-Waiting nämlich zu teuer im Vergleich zu den Extra-Kontextwechseln, die durch ein “sleep” verursacht würden.

Q: Wie funktioniert ein Turnstile in Solaris?
A: Ein Turnstile ist nichts anderes als ein Queue-Lock, das vom Kernel verwaltet wird und die wartenden Threads in Reihenfolge enthält. Zusätzlich zu den so gewonnenen FIFO-Eigenschaften wird Priority-Inversion verhindert, indem die Priorität des Lock-Halters auf die höchste Priorität aller Prozesse in der Queue gesetzt wird.

Q: spontan frage ich mich, ob ihr gute Ressourcen (im Netz, Bücher, Lehrmittel) kennt, die die eingesetzten Konzepte und Zusammenhänge/Eigenheiten in einem bestimmten Betriebssystem, z.B. Windows, übersichtlich darstellen. Meist gelangt man beim Googeln schon auf irgendeine MSDN Seite, die aber gerade mal eine knappe Doku zur gesuchten Methode, im besten Fall ein Beispiel liefert. Mir wären Diagramme und Graphen lieber...
A: Fuer Windows ist MSDN die Authoritaere Resource, fuer Linux/Unix gibt es diverse Buecher, empfehlenswert ist TLPI: The Linux Programming Interface.

Q: S. 20: was versteht man unter einem SMP System, einer SMP Architektur?
SMP = Symmetric Multiprocessing, eine Architektur bei der mehrere, identische Prozessoren, sich einen Memory Bus teilen. Heute sind alle Architekturen SMP. Nennenswerteste Ausnahme: der PlayStation3-”Cell”-Prozessor, der aber der schwierigen Programmierbarkeit nie sein ganzes Potenzial entfaltet hat.

Thema 5, Deadlocks

Q: Seite 37 im PDF: «A deadlock-free solution does not necessarily eliminate the possibility of starvation». Beispiel?
A: Angenommen zwei Prozesse greifen auf eine Resource zu, dann kann es sein das ein Prozess immer Pech hat und nie drankommt, weil der andere Prozess die Resource freigibt und direkt wieder an sich reisst. Das ist Deadlock free, weil ein Prozess Fortschritt macht, aber nicht starvation free weil der erste Prozess nie voran kommt.

Q: With the graphmodel we can see if a configuration of processes may run into a deadlock, but how do we deal with it if it occurs? And how do we recognize it?
A: Recognition: Wait-for graph: If there is a cycle, there is a deadlock. (Detect cycles: Tarjan's strongly connected component algorithm might be a first step (linear time); Donald B. Johnson: "Finding all the elementary circuits of a directed graph")

Recovery (Section 7.7): 1) Kill one or all deadlocked processes. 2) Take resources from some processes -> How can we roll back the affected processes to a safe state? How to make sure not always the same process is affected (and thus starves)?

Better: Prevention: Total order on the resources. Processes have to request resources in this order. Then, always one process will be able to continue. (cf. Dining Philosophers)

Q: Kapitel 7.2: Are there tools to visualize these kind of graphs? (While executing a program, similar to profiling e.g. gprof, ...)
A: There are tools that instrument lock interactions so that deadlocks can be detected, visualizing them should not be difficult. Most tools simply print a list of locks acquired by the processes.