Weiter zu Kapitel2 + Kapitel 3 - Nebenläufigkeit (Concurrency):

ABER: über Größe der Pages nicht viel zu gewinnen -> auch: bei linearen Page Tables können viele Einträge ungültig sein

Multi-Level Page Tables (MLPT): Page Table wird hierarchisch aufgebaut ->Ziel: beseitigen ungültiger Einträge einer lineare Page Table -> für entsprechende Einträge soll kein Speicher alloziert werden

Linear Page Table (LPT): Es gibt Page Table Register (PTBR) |LPT ist selbst in Pages unterteilt, welche in PFs im Speicher liegen |Es gibt im Speicher 256 PTEs (8 Bit VPN) |Bei 4 Byte pro PTE gibt es 16 PTEs, die auf einer Page der LTB liege n| man benötigen 16 Pages, um die LPT im Speicher zu halten | PT hat eine Größe von 1 KB
-> Jetzt Speicherbedarf minimieren: Bisher Linear Page Tables (Nur mit VPN indiziert)

Zweite Idee: 2-Level Page Table: um 2-Level Page Table zu erhalten, wird ursprüngliche Page Table in Blöcke der Größe einer einzelnen Page unterteilt -> falls alle Einträge einer Page der Page Table ungültig sind, dann wird kein Speicher für die (gesamte) Page alloziert

Page Directory (PD): wird zusätzliche Datenstruktur benötigt, die gültige Regionen speichert (Page Directory, PD)
->Darin wird festgehalten, ob eine Page der Page Table gültig ist und wo diese gegebenenfalls zu finden ist
-> Wir erhalten damit hierarchische und dünnbesetzte (sparse) Darstellung der ursprünglichen linearen Page Table:
-> Level 1: Page Directory mit Page Directory Entries (PDEs) |-> Level 2: Page Table mit Page Table Entries (PTEs)

Bemerk: -Page Directory (Lvl1) wird so angelegt, dass es auf Page passt (Auf Page passen 16 Einträge, wodurch man 4 Bit für Page Directory Index (PDI) reservieren muss) ->Weitere 4 Bit werden dann für Page Table Index (PTI) reserviert

Vorteile: -Anstelle der 16 Pages für eine LPT, benötigt die 2-Level PT nur 3 Pages.
-die 2-Level PT hat eine Größe von 192 Byte, während die LPT 1024 Byte benötigt.

Nachteile: -Komplexität der Struktur steigt |-Bei TLB Miss benötigen wir 2 zusätzliche Speicherzugriffe, um die richtigen Informationen in der PT zu erhalten (1x PDE, 1x PTE) |

Einträge des Page Directory: PD besteht aus PDEs und enthält Eintrag für jede Page der Page Table -> PDE enthält mind. ein Valid Bit, sowie die zugehörige PFN für entsprechende Page der Page Table. -> durch diese Struktur müssen benötigte Pages der Page Table nicht zusätzlich alloziert werden (MLPT werden mittels RadixTrees implementiert)

Adressübersetzung für 2-Level PT: Schritte um von virtueller zu Physischer Speicheradresse zu kommen **(Page Walk)**
1. Berechnung der Adresse des PDE: *PDEAddr = PageDirBase + (PDIndex * sizeof(PDE))* | **2**. Wenn PDE.valid, dann berechne PTE Adresse aus PDE und dem Page Table Index: PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE)) | **3**. Wenn PTE.valid, dann wird aus der PFN und dem Offset die PA berechnet:

Mehrere Stufen: Konzept der 2-Level PT lässt sich auf beliebig viele Stufen erweitern -> Frage nach richtiger Anzahl
--> Um dies zu entscheiden, müssen wir die folgenden Eigenschaften beachten:
->Jeder Teil der PT soll auf eine Page passen -> abhängig vom VAS, der Page Size sowie Größe von PTEs + PDEs

Bsp: 30 Bit VAS mit 512 Byte Pages (9 Bit Offset, 21 Bit VPN) -> PTE Größe von 4 Byte: 128 PTE pro Page und 7 Bit Page Table Index |->PDE Größe von 4 Byte: 128 PDE pro Page und 7 Bit Page Directory Index. (3h für PageTable nötig)

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT # VPN ermitteln
(Success, TlbEntry) = TLB_Lookup(VPN) # TLB Lookup
if (Success == True): # TLB Hit
  if (CanAccess(TlbEntry.ProtectBits) == True): # Zugriff erlaubt
    Offset = VirtualAddress & OFFSET_MASK # Offset ermitteln 3)
    PhysAddr = (TlbEntry.PFN << SHIFT) | Offset # PFN + Offset 3)
    Register = AccessMemory(PhysAddr) # Daten laden (Speicherzugriff!)
  else:
    RaiseException(PROTECTION_FAULT) # Zugriff nicht erlaubt
else: # TLB Miss
  PDIndex = (VPN & PD_MASK) >> PD_SHIFT # PD Index ermiteln
  PDEAddr = PDBR + (PDIndex * sizeof(PDE)) # PDE-Adresse ermitteln 1)
  PDE = AccessMemory(PDEAddr) # PDE laden (Speicherzugriff!)
  if (PDE.Valid == False): # PDE nicht valid
    RaiseException(SEGMENTATION_FAULT) # Segmentation Fault
  else: # PDE valid
    PTIndex = (VPN & PT_MASK) >> PT_SHIFT # PT Index ermitteln
    PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE)) # PTE-Adresse 2)
    PTE = AccessMemory(PTEAddr) # PTE laden (Speicherzugriff!)
    if (PTE.Valid == False): # PTE nicht valid
      RaiseException(SEGMENTATION_FAULT)
    elif (CanAccess(PTE.ProtectBits) == False): # Zugriff nicht erlaubt
      RaiseException(PROTECTION_FAULT)
    else: # Zugriff erlaubt
      TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits) # PTE > TLB
      RetryInstruction() # Nochmal ausführen (dann dritter MEM-Access)
```

Ablauf eines Speicherzugriffs:
1. Virtuelle Adresse in DTLB nachschlagen
-Hit: Benutze physische Adresse
-Miss: Virt Adr muss in STLB gesucht werden
2. Virtuelle Adresse in STLB nachschlagen
-Hit: Aktualisiere DTLB und benutze phys Adr
-Miss: VA mittels Page Walk ermittelt werden
3. Page Walk…
--> hängt von Architektur ab, ob auf Hauptspeicher zugegriffen werden kann

->weitere Mglkeit, um Speicherbedarf von Page Tables zu senken:
Inverted Page Tables (IPT): IPT bildet von PFNs auf VPNs ab (vom PAS zu VAS) -> nur eine IPT pro PAS nötig, die von allen Prozessen geteilt wird

-> Als Möglichkeit, um die Suche zu beschleunigen, lassen sich sogenannte Hashed Inverted Page Tables nutzen.

Speicher ist Limitiert: designen ein System, das viele gleichzeitig existierende große Adressräume unterstützt
->wir bauen auf **Speicher Hierachie**: mehrere Stufen mit unters Geschwindigkeit und Größe: Register, Cache(-Level), Hauptspeicher, Festplatte, … -> schnelle Speicher ist klein, großer Speicher ist langsam

Aufg: Wie kann Betriebssystem größere und langsamere Speicherressourcen nutzen, um die Illusion eines (unendlich) großen virtuellen Adressraums aufrechtzuerhalten? -> Wir müssen Daten auslagern! (Swapping)

Swapping: ist das Laden und Auslagern von Daten innerhalb einer Speicherhierarchie (und bei Bedarf geladen)

Swap Space: Ort, welcher explizit für Auslagern von Pages des Page Tables vorgesehen ist
-Ort ist eingeteilt in Blöcke der „richtigen Größe“ (Page Size) -> Present Bit eines PTEs gibt Auskunft, ob sich eine gesuchte Page gerade im Hauptspeicher befindet oder nicht.
-Present Bit in PTE sagt, ob Page im RAM oder Swap. ->Wird Page benötigt und Present Bit nicht gesetzt, wird Page Fault Exception ausgelöst (Page Fault Handler ist zuständig.)

Page Faults in Adressübersetzung:

```
# Page Faults
PTEAddr = PTBR + (VPN * sizeof(PTE)) # PTE Adresse ermitteln
PTE = AccessMemory(PTEAddr) # PTE laden
if (PTE.Valid == False): # PTE nicht valid
RaiseException(SEGMENTATION_FAULT)
else: # PTE valid
if (CanAccess(PTE.ProtectBits) == False): # Zugriff nicht erlaubt
RaiseException(PROTECTION_FAULT)
elif (PTE.Present == True):
TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits) # PFN > TLB
RetryInstruction() # Nochmal
elif (PTE.Present == False): # Page nicht im RAM
RaiseException(PAGE_FAULT) # -> PageFault

# Page Fault Handler
PFN = FindFreePhysicalPage() # Freien Page Frame suchen
if (PFN == -1): # keine Freie Page gefunden
PFN = EvictPage() # Replacement Strategie
DiskRead(PTE.DiskAddr, PFN) # Auf IO warten
PTE.present = True # Jetzt ist Page im RAM
PTE.PFN = PFN # PFN setzen
RetryInstruction() # Nochmal Instruktion
```

Pages ersetzen: irgendwann keine freien Pages mehr und alte Pages müssen ersetzt werden
-> beim Swapping heißen diese Strategien Cache Replacement Policies genannt
-> Ziel: Hit Rate im Cache zu maximieren.

Strategien:
Optimal (Belady): Theoretische Strategie, die zu geringst möglichen Miss Rate führt -> Es wird diejenige Page ersetzt, deren nächste Nutzung am weitesten in der Zukunft liegt.
First In, First Out (FIFO): Verwerfe Page, deren Ladevorgang am längsten zurückliegt (siehe Scheduling aus Kap. 1)
Least-Recently-Used (LRU): Verwerfe Page, deren Verwendung am längsten zurückliegt (TLB)
Random: Verwerfe eine zufällige Page (TLB)

zu LRU-Implementierungen:
-Zugriffe auf Pages im Cache müssen überwacht werden, um berechnen zu können, welche Page ersetztbar (**Accounting**).
-> ist bei großen Anzahl an Pages teuer, sodass es Approximationen gibt (z.B Clock-Algorithmus).
-Worst-Case-Szenarien und Randfälle sollen vermieden werden

Weitere Methoden im Zusammenhang mit Swapping:
-Betriebssystem bringt Pages spekulativ in Hauptspeicher unter Annahme, dass diese bald benötigt werden (**Prefetching**)
-Swapping von Pages in realen Systemen wird oft mittels sogenannter Watermarks erweitert -> Unterschreitet die Anzahl freier Page Frames im Hauptspeicher die Low Watermark, werden automatisch Pages in den Swap Space geschoben, bis die High Watermark erreicht ist.

Kapitel 3: Concurrency

Concurrency: Ziel: Nebenläufigkeit von Aufgaben innerhalb eines Programms (Multi-Threaded-Anwendung). Warum? /-Parallelisierung /-Aufteilen von Rechenaufgab auf mehrere Prozessoren /-kein Blocken von Programmen durch slow IO

Prozesse aufteilen: um Rechenaufgaben verteilen zu können oder versch Teile eines Programms unabhängig voneinder laufen lassen zu können, braucht man Teilprozesse -> Threads

Thread: -ist unabhängiger Agent, der Aufgaben im Namen des Programms ausführt ->funktionieren prinzipiell wie Prozesse, teilen sich aber einen gemeinsamen Speicheradressbereich
-jeder Thread verfügt über einen eigenen Program Counter (PC) und eigene Register |-Ausführen mehrerer Threads auf dem selben Prozessor setzt Context-Switch voraus |-Adressbereich bleibt beim Context-Switch zwischen Threads erhalten, was mehrere Stacks pro Adressbereich zur Folge hat |-Zusätzlich zum Process Control Block (PCB) wird ein weiterer Thread Control Block (TCB) pro Thread zum Speichern des Thread-Zustandes benötigt

Bemerk: gemeinsamer Heap aber separate Stacks (Thread-Local Storage) pro Thread

Problem: Einzelner Stack kann einfach vom Ende des Adressbereichs negativ wachsen, mehrere Stacks nicht.

in Praxis: selten ein Problem da Stacks i.d.R. klein sind ->Ausnahme: Algorithmen, welche viel von Rekursionen Gebrauch machen ->Außerdem haben auch noch Speichervirtualisierung (siehe Kapitel 2).

Bsp: Ausführen mehrerer Threads: 3 Varianten, siehe Vorlesungsfolien

Kernproblem: Scheduling ->wenn Threads erstellt und unterschiedli ablaufen, dann sind abhängig davon wie Scheduler entscheidet sie auszuführen -> auf Programmebene keine Kontroller über Scheduler
->es gibt viele unterschiedl und häufiger falsche als richtige Ergebnisse bei mehreren unabhängigen Durchläufen
->Operationen in ***kritischen Abschnitten*** sind nicht atomar, somit beeinflusste Ergebnisse vom Scheduling

Anmerk: **kritischer Abschnitt:** ist Teil des Codes, der auf gemeinsame Ressource zugreift (zB: Variable oder Datenstruktur)

Atomar: steht für ”als ganzes” oder ”alles oder nichts” ->Atomare Operationen können nicht mittendrin unterbrochen werden (Interrupt) ->Findet ein Interrupt statt, laufen sie entweder gar nicht oder vollständig vorher ab.

Problem: **Race Condition**: Ergebnisse hängen von zeitlicher Ausführung des Codes ab (bevor zurückgeschrieben wird bereits anderer Thread geladen) -> Ergebnis dann auch nicht deterministisch.

Lösung: **Mutual Exclusion**: garantiert, dass bei Ausführung eines kritischen Abschnitts durch Thread, andere Threads daran gehindert werden den Abschnitt ebenfalls auszuführen (RaceConditions verhindert + determinstAusgaben garantiert)

Problem: um Atomare Operationen für alle kritischen Abschnitte zu definieren, bräuchte man für jede parallele Operation eine beliebigen Datenstruktur (Listen, Bäume, Keller, usw) -> unrealistisch, teuer und unflexibel

Lösung: **Synchronisierungsprimitive**: **1**-braucht nur wenige Hardware-Instruktionen auf die allgemeingültige Synchrprimitive aufbauen | **2**-mithilfe Hardware, zsm mit Betriebssystem, kann man in Multi-Thread-Programme schreiben, die ***kritische Abschnitte*** synchronisiert und kontrolliert ausführen kann

Fragen: Welche Unterstützung von HW und I/O nötig um nützliche Synchroprimitive zu erstellen? | Wie Primitive korrekt und effizient erstellen? |Wie können Programme diese nutzen, um die gewünschten Ergebnisse zu erzielen?
->wie auf Threads warten? ->Welche Mechanismen benötigt um Unterstützung von Schlaf-Wach-Interaktionen?

Bsp: die Ausführung des kritischen Abschnitts mit Lock synchronisieren. Dazu verschließt man Abschnitt, indem man Lock setzt -> Nach dem kritischen Abschnitt wird Lock wieder deaktiviert

Schlossvariable/Sperre (Lock): -ist einfache Variable, deren Nutzung zunächst Deklaration voraussetzt (zB. mutex) -> hält den Zustand des Locks zu jedem beliebigen Zeitpunkt |-Lock ist entweder verfügbar (available, unlocked oder free), so dass kein Thread den Lock hält, oder gesperrt (acquired, locked oderr held), so dass genau ein Thread Lock hält.

Verhaltensregeln: **1**-Mit Aufruf von lock() versucht ein Thread (T1) Lock zu beanspruchen ->Wenn kein anderer Thread den Lock hält (frei ist), wird T1 Lock erhalten und kritischen Abschnitt ausführen können -> dieser Thread ist Eigentümer des Locks | **2**-Wenn weiterer Thread (T2) lock() aufruft, wird Routine solange nicht terminieren wie Lock von T1 gehalten wird ->Zugriff auf kritischen Abschnitt verhindert | **3**-Sobald Eigentümer des Locks (T1) unlock() aufruft, ist Lock wieder verfügb und der lock()-Aufruf von T2 terminiert (T2 erhält den Lock)

Anforderungen von OS und I/O: -***Korrektheit***: Lock muss Mutual Exclusion garantieren, also verhindern, dass mehrere Threads gleichzeitig den kritischen Abschnitt ausführen |-***Fairness***: Jeder Thread soll faire Chance auf Lock haben | -***Performance***: zeitliche Overhead für die Verwaltung sollte gering sein -> vers Fälle berücksichtigen (1, mehre Prozessor)

Idee um Mutual Exclusion zu ermöglichen: **Locks mit Interrupts kontrollieren**: lock() deaktiviert Interrupts; unlock() aktiviert
->Vorteile: -einfache Umsetzung |-Korrektheit von Mutual Exclusion ist für einen Prozessor gewährleiste
->Nachteile: -Priviligierte Operation nötig: Interrupt ausschalten (Hoffnung: kein Missbrauch) |-keine Korrektheit für mehrere Prozessoren |-Gefahr, dass Interrupts verloren gehen |-Ein-/Auschalten von Interrupts langsam

Nächste Idee: **Naiver Spin Lock**: Marker Variable (Flag); Instruktionen Load/Store und Warteschleife (Spin-Lock)

```
typedef struct __lock_t { int flag; } lock_t; // Datenstruktur
void init(lock_t *mutex) { mutex->flag = 0; } // Initialisieren
void lock(lock_t *mutex) {
  while (mutex->flag == 1) // Wenn danach Interrupt ausgeführt wird
  // wird Flag 2x gesetzt -> Mutual Exclusion nicht gewährleistet!
  ; // spin-wait (do nothing)
  mutex->flag = 1; }
void unlock(lock_t *mutex) { mutex->flag = 0; } // Lock freigeben
```

Spin Lock mit HW: Einfachste Variante: Test-And-Set-Instruktion. Dabei wird Unterstützung der HW genutzt:

```
int TestAndSet(int *old_ptr, int _new) {
  int old = *old_ptr; // Alten Wert auslesen
  *old_ptr = _new; // Neuen Wert in old_ptr speichern
  return old; // Alten Wert zurückgeben
} In lock-Funktion: while (TestAndSet(&lock->flag, 1) == 1) ; // spin-wait
```

Bemerk: -Rückgabe des alten Wertes und setzen eines neuen Wertes ist als atomare Operation implementiert |-Test-And-Set ermöglichest es uns Wert einer Speicheradresse zu überprüfen (Rückgabewert) und gleichzeitig zu setzen |-Damit erhalten wir einen funktionierenden Spin Lock |-Auf SPARC wird dieser Lock als Load/Store Unsigned Byte Instruction (ldstub) und auf x86 als Locked Atomic Exchange (xchg) unterstützt.

Test auf Korrektheit bezüglich Mutual Exclusion:

Fall1: **1**-Ein Thread (T1) ruft lock() auf. Kein anderer Thread hält derzeit den Lock, flag ist also 0 | **2**-T1 ruft TestAndSet (flag,1) auf und erhält für flag den Wert 0 zurück. T1 verlässt Schleife sofort, hat dabei flag (atomar) auf 1 gesetzt und somit den Lock beansprucht | **3**-T1 tritt in den kritischen Abschnitt ein | **4**-Wenn T1 kritischen Abschnitt beendet hat ruft er unlock() auf um flag wieder auf 0 zu setzen und so den Lock freizugeben.

Fall2: **1**-Ein Thread (T1) hält Lock; weiterer Thread (T2) ruft lock() und damit auch TestAndSet(flag, 1) auf | **2**-T2 erhält für flag den Wert 1 zurück und setzt ihn dabei wieder auf 1 (atomar) |**3**-Solange Lock von T1 gehalten wird, wartet damit T2 in der Schleife |**4**-Wenn flag von T1 wieder auf 0 gesetzt wird, erhält T2 im nächsten Aufruf von TestAndSet() 0 zurück, und setzt dabei den flag (atomar) auf 1 (Lock beansprucht) | 5-T2 tritt in den kritischen Abschnitt ein.

Evaluation von Spin Locks: (Fairness; Performance CPU, CPUs)

Fairness: Einfache Spin Locks bieten keine Fairness-Garantie u können zum Aushungern einzelner Threads führen. Wartender Thread kann im Wettbewerb mit anderen Threads eventuell für immer warten, falls er nicht (zufällig) im richtigen Moment den Lock beansprucht. Keine Fairness!

Performance (Einzelne CPU): -teuer! Thread hält den Lock und wird innerhalb eines kritischen Abschnitts unterbrochen. Der Scheduler wird anderen konkurierenden Thread laufen lassen, die alle versuchen Lock zu beanspruchen. Alle Threads verschwenden Prozessorzeit! Schlecht!

Performance (Mehrere CPUs): fkt gut, wenn Anzahl der Threads ungefähr der Anzahl der Prozessoren entspricht. Allerdings gibt es prinzipiell das gleiche Warteproblem und Verschwendung von Rechenzeit. s.o.

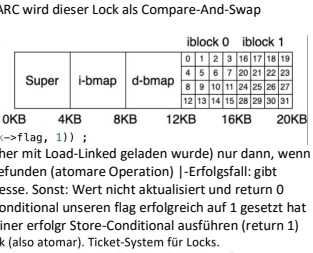
Compare-And-Swap: Ähnlich wie Test-And-Set aber unterstützt Lock-free Synchronisation

```
int CompareAndSwap(int *ptr, int expected, int _new) {
  int original = *ptr;
  if (original == expected) *ptr = _new; // Wenn Wert = erwarter Wert => Überschreiben
  return original;
} In Lock(): while (CompareAndSwap(&lock->flag, 0, 1) == 1) ; // spin-wait
```

Bemerk: -Test-And-Set und Compare-And-Swap erfüllen gleiche Funktion (gleichen Vor- und Nachteile) |-Compare-And-Swap mächtigere Variante ->Lock-free Synchronization umsetzbar |-Auf SPARC wird dieser Lock als Compare-And-Swap unterstützt (bei x86: Compare-And-Exchange)

LoadLinked: Wieder Nutzen von Load/Store (diesmal mit HW-Unterstützung):

```
int LoadLinked(int *ptr) { return *ptr; }
int StoreConditional(int *ptr, int value) {
  if (no update to *ptr since LoadLinked to this address) {
  *ptr = value;
  return 1; // success!
  } else return 0; // failed to update
} Lock(): while (LoadLinked(&lock->flag) || !StoreConditional(&lock->flag, 1)) ;
```

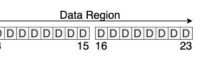


Bemerk: -Store-Conditional aktualisiert Wert an Speicheradresse (die vorher mit Load-Linked geladen wurde) nur dann, wenn keine zwischenzeitliche Speicherung an dieser Speicheradresse stattgefunden (atomare Operation) |-Erfolgsfall: gibt Store-Conditional Wert 1 zurück und aktualisiert Wert an Speicheradresse. Sonst: Wert nicht aktualisiert und return 0

Bauen von Lock: warten (spin) solange bis flag (wieder) 0 ist + bis Store-Conditional unseren flag erfolgreich auf 1 gesetzt hat ->if 2 Threads durch Interrupts Load-Linked ausführen + 0 erhalten; nur einer erfolgr Store-Conditional ausführen (return 1)

Fetch-And-Add: Bisher nicht fair. FaA erhöht um 1, gibt dann alten Wert zurück (also atomar). Ticket-System für Locks.
Statt Flag Ticketzähler (ticket) und Rundenzähler (turn). FetchAndAdd->Ticket ziehen. Dann warten, bis man dran ist. Ist fair.

```
int FetchAndAdd(int *ptr) { int old = *ptr; *ptr = old + 1; return old; }
typedef struct lock_t { int ticket; int turn; } lock_t;
void lock_init(lock_t *lock) { lock->ticket = 0; lock->turn = 0; }
void lock(lock_t *lock) {
  int myturn = FetchAndAdd(&lock->ticket);
  while (lock->turn != myturn) ; // Warten, bis man dran ist }
void unlock(lock_t *lock) { lock->turn = lock->turn + 1; } // Freigeben mit Increment
```



Bemerk: -Statt Flag gibt es Ticketzähler (ticket) und Rundenzähler (turn) |-Mittels Fetch-And-Add wird Ticket gezogen |-es wird gewartet bis turn gleich myTurn (eigene Ticketnummer) ist |-nach kritischen Abschnitts wird turn erhöht, um Lock wieder freizugeben |->wird sichergestellt, dass alle Threads Möglichkeit haben, den Lock zu erhalten (Ticket-System)

ZwStand: durch Tickets faire/korrekte Implementierung für Spin Locks ->Problem noch bei Performance -> Lsg: yield()

Einfach Aufgeben mittels yield(): Thread in einem von drei Zuständen befindend (Running, Ready oder Blocked)
-> yield() bewirkt, dass aufrufende Thread vom Zustand Running in den Zustand Ready versetzt wird
-> Damit wird ein anderer Thread in den Zustand Running befördert.

->Einfacher Lock mittels Test-And-Set und yield() -> Code … (• Korrektheit: Ja!; • Fairness: Nein!; • Performance: Unklar!)

Evaluation der Performance:

Performance (2 Threads): guter Yield-Ansatz. Wenn Thread zufällig lock() aufruft aber Lock bereits gehalten wird, dann gibt er Prozessor frei. Andere Thread kann laufen und seinen kritischen Abschnitt beenden.

Performance (100 Threads): Wenn Thread (T1) den Lock erwirbt und vor Freigabe unterbrochen wird, rufen die anderen 99 Threads u.U. jeweils lock() auf und geben Prozessor sofort frei. Bei Round-Robin-Strategie, wird jeder der 99 Threads „Run-and-Yield-Muster“ ausführen, bevor T1 erneut an der Reihe ist. Kosten für Context-Switch können beträchtlich sein

Locks mit Warteschlangen: -Threads schlafen legen |-**park()** und **unpark()**, die Thread schlafen legen bzw. wieder aufwecken können -> durch einer Warteschlange wird Reihenfolge festgelegt, sodass wir Einfluss auf die Ablaufplanung haben.

Implementierung der Locks: - einmal im Code, siehe Folien….
-Variable guard dient als einfacher Spin Lock um nicht-atomare lock()-Anfrage zu synchronisieren |-Wenn Lock frei ist (flag=0), dann wird er beansprucht |-Ist Lock blockiert (flag=1), dann wird Thread in die Warteschlange eingereiht, guard wieder freigeben, und dann der Thread schlafen gelegt (park).

unlock()-Methode in Code: -Variable **guard** dient als einfacher Spin Lock um nicht-atomare lock()-Anfrage zu synchronisieren
-Wenn queue leer ist, dann wird der Lock freigegeben (flag=0) |-Ansonsten wird nächste Thread aus queue geholt und mit unpark() aufgeweckt (ohne Lock freizugeben)

werden (interrupt) ->Findet ein Interrupt statt, laufen sie entweder gar nicht oder vollständig vorher ab.