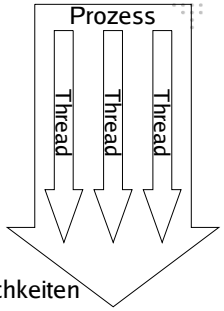
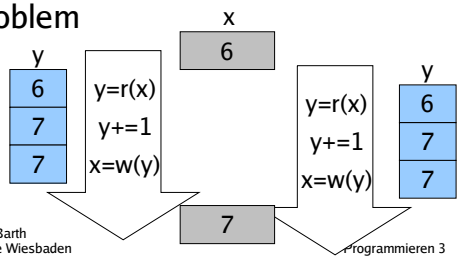
Thread-Programmierung

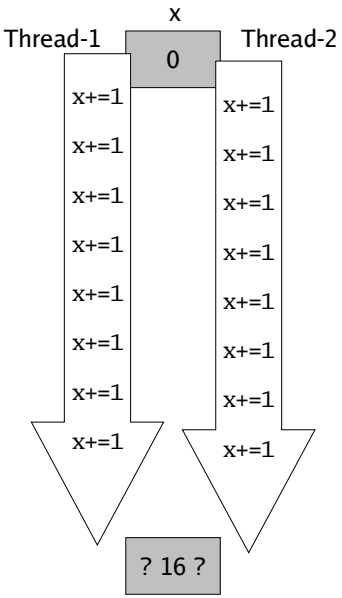


- Threads, Nebenläufigkeit
  - Mehrere Ausführungsstränge gleichzeitig bei gemeinsamen Daten/Ressourcen
  - Leichtgewichtige Realisierung auf Systemebene, effizient
  - Komplex, mächtig bei Verwendung
  - In modernen Programmierumgebungen unterstützt
    - In C/C++ Integration von Bibliotheken/Betriebssystemmöglichkeiten
    - In Java Teil der Sprache, in Python ähnlich (API) wie Java
  - Prozesse: Separaten Ausführungsstränge und separate Daten/lokale Ressourcen, schwergewichtige Realisierung, hoher Aufwand, einfach zu verwenden (nicht sichtbar)
- Einsatzgebiete
  - Kein Warten bei IO-Operationen, Überbrücken Latenz Netzwerkkomm., z.B. GUI-Programmierung, UI „friert nicht mehr ein“
  - Nutzung von Mehrkernarchitekturen, Parallele Algorithmen, Verteilen der Arbeitslast auf mehrere Kerne
- Hauptproblem: Datensynchronisation

Synchronisation

- Beispiel
  - Eine (globale) Variable x mit Initialwert 0
  - Zwei Threads, die (gemeinsame) Variable x immer um 1 erhöhen
- Genauer
  - x += 1 entspricht
    read x in y
    y= y+1 berechnen
    write y in x
  - y ist (thread-)lokale Variable, ein separates y je Thread
- Problem





Lösungsansatz: Sperren

Thread-Programmierung in Python

- Modul threading
  - Von Klasse Thread erben
  - run-Methode überschreiben
  - Anweisungen in run-Methode bilden separaten Ausführungsstrang
  - Instanzvariablen und lokale Variablen in run sind lokal zum Thread

```
from threading import Thread

x = 0

class Incer(Thread):
    def __init__(self, runs):
        Thread.__init__(self)
        self.runs = runs
    def run(self):
        global x
        for i in range(self.runs):
            x += 1
```

- Beispiel: Inkrementieren von globalem x
- Ausführen
  - Instanz erzeugen
  - start-Methode (nicht run)
    - Kehrt sofort zurück
    - Startet neuen unabhängigen Ausführungsstrang
  - join-Methode wartet bis Thread fertig, Ende von run

```
if __name__ == '__main__':
  runs = 10**5
  inc1 = Incer(runs)
  inc2 = Incer(runs)
  inc1.start()
  inc2.start()
  inc1.join()
  inc2.join()
  if x != 2*runs:
    print "Ooopppss", x, "statt", 2*runs
```

Ooopppss 119274
statt 200000

Threads – Kontrolle Ablauf

- Es gibt keine Möglichkeit Threads von außen zu unterbrechen/stoppen
  - Es **kann** keine solche Möglichkeit geben, die immer funktioniert
  - Der Anwendungsentwickler muss entsprechende Möglichkeiten vorsehen
- Ansatz
  - Spezielle Methoden zum Setzen von Variablen
  - Periodische Überprüfung dieser Variablen
  - Kein Synchronisationsproblem, da ein Thread nur liest, der andere nur schreibt (und Bool immer komplett geschrieben wird)

```
from threading import Thread
from time import sleep

x = 0

class Incer(Thread):
    def __init__(self):
        Thread.__init__(self)
        self.weiter = True
    def run(self):
        global x
        while self.weiter:
            x += 1
            sleep(1)
            print "x", x
    def anhalten(self):
        self.weiter = False
```

Eine Sekunde schlafen

- Beispiel Ausgabe
  - isAlive, läuft Thread noch?
  - join verwenden

```
x 1
x 2
x 3
Lebt? True
x 4
Lebt? False
```

```
inc = Incer()
inc.start()
sleep(3)
inc.anhalten()
print "Lebt?", inc.isAlive()
inc.join()
print "Lebt?", inc.isAlive()
```

Threads – Ende erreicht?

- Niemals aktiv warten!
  - Volle CPU-Last mit Nichtstun/Warten verschwendet
  - Nie, Nie, Nie

```
inc = Incer()
inc.start()
while inc.isAlive():
    pass
# weiter nach Thread-Ende
```

- Thread.join ok
  - Wartet darauf, dass ein Thread beendet wird
  - Aktuell laufender Thread hält, verschwendet aber keine Prozessorzeit
  - Nachteil:
    - Aktueller Thread läuft nicht weiter bei Aufruf von inc.join()
    - Falls „sinnvolles“ lange dauert, dann wird nicht gleich reagiert sobald Thread fertig

```
inc = Incer()
inc.start()
# was sinnvolles
inc.join()
# Ergebnis des Threads
# verwenden
```

- Meist will man nicht warten, sondern nur etwas direkt nach Beendigung des Threads tun
  - Lösung: Verarbeitung noch im Berechnungs-Thread erledigen, „Callback“

Aktion nach Ende Thread

```
def prim(x): # Berechnung
    return [t for t in range(2, x) if x%t == 0] == []
def printPrim(ite, prim): # Verwendung
    print "die %dte Primzahl ist %d" % (ite, prim)
```

Funktion, die Ergebnis verarbeitet

```
class Langlaeufer(Thread):
    def __init__(self, iteprim):
        Thread.__init__(self)
         self.iteprim = iteprim
    def run(self):
        p = 1
        while self.iteprim:
            p += 1
            while (not prim(p)):
                p += 1
            self.iteprim -= 1
        self.prim = p
    def getPrim(self):
        return self.prim
```

NICHT SO!

```
ite = 1000
t = Langlaeufer(ite)
t.start()
# was anderes sinnvolles
t.join()
printPrim(ite, t.getPrim())
```

```
class Langlaeufer(Thread):
    def __init__(self, iteprim, func):
        Thread.__init__(self)
        self.iteprim = iteprim
        self.func = func
    def run(self):
        p = 1
        while self.iteprim:
            p += 1
            while (not prim(p)):
                p += 1
            self.iteprim -= 1
        self.func(p)
```

Funktions-aufruf mit Ergebnis

SONDERN SO

```
ite = 1000
verwende = lambda x: printPrim(ite, x)
t = Langlaeufer(ite, verwende)
t.start()
# was anderes sinnvolles
```

```
die 1000te Primzahl ist 7919
```


Synchronisation mit Sperren, Lock

- Kritischer Bereich
  - Code-Abschnitt, der nur von einem Thread gleichzeitig durchlaufen werden soll
  - Meist bei Lese/Schreib-Operation auf gemeinsame Ressource
  - Im Beispiel: x += 1
- Sperr-Objekt
  - Bietet sichere Sperre, atomare Operationen zur Sperrverwaltung
  - acquire: Belegt Sperre, falls Sperre nicht verfügbar wird aktueller Thread angehalten (schlafen gelegt)
  - release: Gibt belegte Sperre frei, muss belegt sein
  - acquire/release Block umschließt kritischen Bereich
- Sperren ist (relativ) aufwendig
  - Sperren nur wenn notwendig, so kurz wie möglich
  - Sperren ist wichtig, besser korrekte als falsche Software

```
from threading import Thread, Lock

x = 0
lock = Lock()

class Incer(Thread):
    def __init__(self, runs):
        Thread.__init__(self)
        self.runs = runs
    def run(self):
        global x
        for i in range(self.runs):
            lock.acquire()
            x += 1
            lock.release()
```

Ein Lock-Objekt für alle Threads

Gesicherter kritischer Bereich

Kritischer Bereich

Zwei Threads
1 Million Durchläufe:
Ohne Sperre: 0,7 s
Mit Sperre: 3,7 s

Lock, RLock

- Lock – Sperrobjekt, exklusiv
  - acquire(blocked=True):
    - Sperre belegen wenn verfügbar und True zurück geben
    - Wenn Sperre nicht verfügbar Thread blockieren (schlafen legen)
  - acquire(blocked=False):
    - Sperre belegen wenn verfügbar und True zurück geben
    - Wenn Sperre schon belegt, dann False zurückgeben ohne Sperre zu belegen
      - Erlaubt auf belegte Sperren zu reagieren
  - release()
    - Gibt belegten Lock frei
    - Ausnahme falls Lock nicht belegt
- RLock – Sperrobjekt, reentrant
  - Wie Lock
  - Erlaubt verschachtelte acquire/release Aufrufe
    - Bei Lock wäre lock.acquire(), lock.acquire() ein Deadlock
    - Bei RLock wird ein Zähler erhöht
  - Verwendung: Aufruf von sicheren Methoden in sicherer Methode (d.h. wenn man Lock schon hat) ohne Anstrengungen

Sichere Objekte

- Bisher Konventionen
  - Globale Objekte x und lock
  - Konvention, dass jeder, der auf x zugreift diesen Bereich mit lock sperren muss
  - Fehleranfällig (bei Annahme nicht idealer Entwickler)
- Besser Kapselung
  - Eigene Klasse mit sicheren Zugriffsmethoden
    - In Java synchronized Methoden
  - Lokales Sperrobjekt in Instanz der Klasse
  - Typisches Pattern bei value, sonst release schwer
    - with-Statement ab 2.6

```
class Counter(object):
    def __init__(self):
        self.count = 0
        self.lock = Lock()
    def inc(self):
        self.lock.acquire()
        self.count += 1
        self.lock.release()
    def value(self):
        self.lock.acquire()
        ret = self.count
        self.lock.release()
        return ret
```

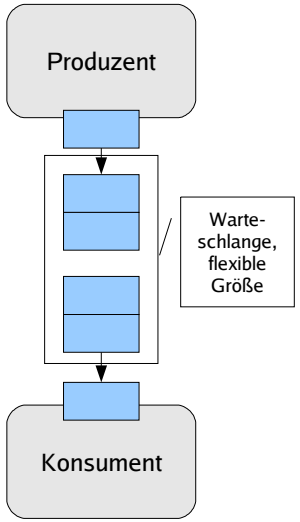
```
class Incer(Thread):
    def __init__(self, counter, runs):
        Thread.__init__(self)
        self.counter = counter
        self.runs = runs
    def run(self):
        while self.runs:
            self.counter.inc()
            self.runs -= 1
```


Besonderheiten Threads in Python

- GIL in CPython Implementierung
  - Global Interpreter Lock
    - Nur ein Python-Thread kann zu einem gegebenen Zeitpunkt auf Python-Strukturen zugreifen
    - Nur ein Python-Thread läuft zu jedem gegebenen Zeitpunkt
  - Keine effektive Nutzung von Mehrkernsystemen möglich
  - Trotzdem sinnvolle Nutzung von Wartezeiten/ bei Langläufern
    - GIL wird bei Warten auf externe Ereignisse (IO) losgelassen
  - Thread-Scheduling von Laufzeitumgebung
    - Alle 100 Bytecode-Anweisungen
    - sys.setcheckinterval
  - Alternative ab Python 2.6: multiprocessing
    - Arbeitet mit mehreren Python-Prozessen
    - Effektive Nutzung von Mehrkernsystemen
- Basiert auf thread-Modul
  - Low-Level API, für die meisten Umgebungen verfügbar
  - Empfehlung: threading verwenden

Producer/Consumer

- Situation
  - Ein Produzent erzeugt Werte
  - Ein Konsument verwendet Werte
  - Beide können unabhängig voneinander laufen
  - Problem:
    - Konsument muss warten bis Produzent fertig ist
    - Produzent muss warten bis Konsument fertig ist
- Warteschlange
  - Entkopplung Produzent und Konsument
  - Produzent legt fertige Werte in Warteschlange ab
  - Konsument holt sich Werte aus der Warteschlange
  - Warteschlange ist gemeinsames Objekt, Zugriffe auf gemeinsames Objekt müssen synchronisiert werden
    - Bei leerer Warteschlange muss Konsument warten
    - Bei voller Warteschlange (begrenzte Größe) muss Produzent warten
    - Einfügen/Entfernen nur einer gleichzeitig (interne Strukturen Warteschlange)



Prod/Cons

- Warteschlange Liste
- Produziert 1,2,3...
- Konsument berechnet Summe

```
class Producer(Thread):
  def __init__ (self, anzahl, queue, queue_lock):
    Thread.__init__(self)
    self.anzahl = anzahl
    self.queue = queue
    self.queue_lock = queue_lock
  def run(self):
    for i in range(1, self.anzahl+1):
      time.sleep(random.random()/100.)
      self.queue_lock.acquire()
      self.queue.insert(0, i)
      self.queue_lock.release()
```

Endlich viele Zahlen

Zeit lassen, damit es spannend wird

```
class Consumer(Thread):
  def __init__(self, queue, queue_lock):
    Thread.__init__(self)
    self.queue = queue
    self.queue_lock = queue_lock
    self.summe = 0
    self.misses = 0   # wie oft nix gekriegt
    self.done = False # Hinweis stoppen
  def run(self):
    while not self.done or self.queue:
      value = None
      self.queue_lock.acquire()
      if self.queue:
        value = self.queue.pop()
      self.queue_lock.release()
      if value:
        self.summe += value
      else:
        time.sleep(random.random()/100.)
        self.misses += 1
```

leer räumen

Wenn leer, dann Prod. aufholen lassen. Darf nicht Bedeutung des Programms ändern.

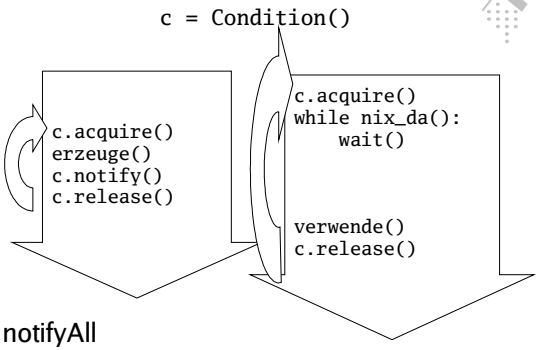
Warteschlange ist Liste. Kapseln mit lock möglich

```
queue = []
lock = Lock()
producer=Producer(anz,queue,lock)
consumer=Consumer(queue,lock)
producer.start()
consumer.start()
producer.join()
consumer.done = True
consumer.join()
```

Sobald Produzent fertig, dies Konsument sagen

Typischer Ablauf bei Anzahl 100: 88 misses

wait/notify



- Condition
  - Klasse von threading
  - Hält intern ein Lock, kann diesen beim Initialisieren übergeben bekommen
  - Implementiert Lock-Interface, u.a. acquire und release
  - Weitere Methoden wait, notify, notifyAll
- wait()
  - Lässt Lock los (Lock muss belegt sein), legt aktuellen Thread schlafen
  - Kann erst weiterlaufen, wenn auf Condition notify oder notifyAll gerufen
- notify()
  - Lock muss belegt sein
  - Informiert einen der wartenden Threads, dass es weiter gehen könnte
  - Programmierer lässt danach lock meist los (release)
- notifyAll(): wie notify, nur werden alle Threads informiert
- Jedes Java Objekt implementiert wait, notify, notifyAll

Producer/Consumer mit wait/notify

- Problem vorher
  - Misses, Aktives Warten
- Lösung
  - Schlafen legen, falls nichts verfügbar
  - Aufwecken falls etwas verfügbar
- Anmerkungen
  - Zwei Konsumenten, jeder versucht zu lesen
  - Queue künstlich auf Länge 1 beschränkt
  - Nicht sicher, dass immer abwechselnd gerufen wird, Problem des Verhungerns
  - notifyAll, damit beide beendet werden

```
queue = []
cond = Condition()
```

```
class P(Thread):
  def __init__(self):
    Thread.__init__(self)
    self.do = True
  def run(self):
    i = 0
    while self.do:
      cond.acquire()
      if not queue:
        queue.append(i)
      cond.notify()
      cond.release()
      i += 1
      sleep(0.3)
    cond.acquire();
    cond.notifyAll()
    cond.release()
```

Max Länge 1

Langsamer

Alle wecken

```
class C(Thread):
  def __init__(self, id):
    Thread.__init__(self)
    self.do = True
    self.id = id
  def run(self):
    summe = 0
    while self.do:
      cond.acquire()
      while not queue and self.do:
        cond.wait()
      val = None
      if queue:
        val = queue.pop()
      cond.release()
      if val:
        summe += val
print self.id, summe
```

wie pollen

aber schlafen bis wecken

Am Ende None

```
p = P(); p.start()
c1 = C(1); c1.start()
c2 = C(2); c2.start()
sleep(4)
p.do = 0
c1.do, c2.do = 0,0
p.join()
c1.join(); c2.join()
```

2 42
1 49

Queue

- Datenstrukturen für Thread-Prog. Queue
  - Klasse Queue, im Modul Queue
  - Warteschlange für konkurrierenden Zugriff
    - Konkurrierende Schreiber und Leser
- Queue
  - put(e): Fügt Element e ein
  - get(block=True, timeout=None): Holt Element raus und gibt es zurück
    - block=True: Blockiert, falls kein Element verfügbar ist
    - timeout = <secs>, wartet maximal <secs> Sekunden bis Element verfügbar ist, wirft dann Empty-Ausnahme
- Wenn möglich dedizierte Datenstrukturen verwenden
  - Für Java: java.util.concurrent

```
from Queue import Queue, Empty
queue = Queue()
```

```
class P(Thread):
  def __init__(self):
    Thread.__init__(self)
    self.do = True
  def run(self):
    i = 0
    while self.do:
      queue.put(i)
      i += 1
      sleep(0.3)
```

```
class Q(Thread):
  def __init__(self, id):
    Thread.__init__(self)
    self.do = True
    self.id = id
  def run(self):
    summe = 0
    while self.do:
      try:
        val = queue.get(True, 1)
        summe += val
      except Empty:
        pass
    print self.id, summe
```

Alle Sekunde prüfen, ob es noch weiter gehen soll

Ende