1// just starting this other people are welcome to make any changes ideally the notes will follow the structure of the lecture slides, also i'm hoping to get a formula's section going of key things to
// i probably won't be able to make too much of this before the exam as i'm planning on doing past papers

learn
Amdahl's Law

$$\therefore S_{overall} = \frac{1}{(1-F) + \frac{F}{S_{opt}}}$$

Efficiency

$$E = \frac{S_{overall}}{P}$$

Cache Calculation formulas
Index = $Log_2(Number of Lines)$

Offset=$Log_2(LineSize)$

Tag = Number of bits - Index - Offset
Number of lines = Size of cache / Line size

Blocked Multithreading

$$N_{sat} = \frac{R+L}{R+C}$$

$$U_{sat} = \frac{R}{R+C}$$

where
N_sat = Saturation point of num. threads while performance is increasing linearly
U_sat = Utilisation of processors after max saturation has been reached
R = Busy time between context switches
C = Time taken for context switch
L = Latency

Types of parallelism

ILP - Instruction
DLP - Data
TLP- Thread/Task
TLP - Transaction

Superscalar
Vector
SIMD
GPU

NUMA Vs UMA

Memory fence

https://code.google.com/p/hatch/wiki/SplitTransaction

Miss Types
COLD
Capacity
Conflict

W.R.T with respect to

Vector - difficult to roll back interrupts
compiler needs to loop vectorize
not very efficient for non vectorized

Trace instructions caches - are used for predictions in program order

cache calculations
http://cseweb.ucsd.edu/classes/su07/cse141/cache-handout.pdf

Hi, Would you plz add some contents about synchronization and multithreading?thx -_-
Yea i will try but if you're really stuck the PPLS course may have good notes about synchronization specifically algorithms for CS and Mutual exclusion -

From PPLS
Mutual Exclusion *is more like anti-synchronization! We want to prevent two or more threads from being active concurrently for some period, because their actions may interfere incorrectly. For example, we might require updates to a shared counter (e.g., count++) to execute with mutual exclusion.*

Condition Synchronization *occurs when we want to delay an action until some condition (on*

the shared variables such as in producer-consumer, or with respect to the progress of other threads such as in a barrier) becomes true.

Properties:

- ***Mutual Exclusion.*** *At most one thread is executing the critical section at a time.*
- ***Absence of Deadlock (or Livelock)****. If two or more threads are trying to enter the critical section, at least one succeeds.*
- ***Absence of Unnecessary Delay****. If a thread is trying to enter its critical section and the other threads are executing their non-critical sections, or have terminated, the first thread is not prevented from entering its critical section.*
- ***Eventual Entry (or No Starvation)****. A thread that is attempting to enter its critical section will eventually succeed.*

Lock *- allows only one thread to enter the part that is locked and the lock is not shared with any other processes*

- ***lock****() and* ***unlock****()*
- *Spinlock*
- *Bakery Algorithm*

Barrier *- A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.*

- *Counter barrier*
- *Sense reversing barrier*
- *Symmetric barriers*
- *Dissemination Barriers*

Semaphore *- does the same as a lock but allows x number of threads to enter, has hardware/OS support*

- ***P****() and* ***V****()*
  s = 1; // initially
  P(s): <await (s>0) s=s-1;>
  V(s): <s=s+1;>
- ***binary*** *- a semaphore whose usage is organised to only ever take the value 0 or 1*
- ***counting semaphore*** *- a semaphore whose value is counting availability of some resource*

Monitor *- an object or module intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared to reasoning about parallel code that updates a data structure.*

- ***wait*** *and* ***signal***

- *The key difference (between monitors and semaphores) is that signal() on a condition variable is not "remembered" in the way that V() on a semaphore is. If no threads are waiting, then a signal() is "lost" or "forgotten", whereas a V() will allow a subsequent P() to proceed.*

Scalability

Increasing the resources times x yields close to x time better performance

- Resources usually means processors but can mean mem./interconnection bandwidth/mem. bandwidth
- Usually means with x times more processors we can get ~x time better performance.

Types of scalability

Time Constrained

- Execution time is kept fixed
- Goal is to increase the amount of data processed
- Increase number of processors and available mem.
- Speedup is then defined by:

$$Speedup_{tc} = \frac{work(p\ processors)}{work(1\ processor)}$$

Problem Constrained

- Problem size is kept fixed
- Goal is to decrease execution time
- Increase number of processors and avail. mem.
- Speedup is defined by:

$$Speedup_{pc} = \frac{time(1\ processor)}{time(p\ processors)}$$