Systems Programming

Spring 2023

14주차

- 기말고사 안내

- Q&A
  - 18-optimization
  - 19-optimization-cache
  - 20-parallelism

기말고사

- 범위
  - 14-concurrent-programming ~ 20-parallelism
  - 강의자료 및 Q&A Session Materials
  - 실습내용 (Kernel Lab)

< 기말고사 범위 >

- 11주차 (5/18)
  - 14-concurrent-programming
  - 15-io-model
- 12주차 (5/25)
  - 16-sync-basic
  - 17-sync-advanced
  - 17-sync-supplement
- ~~13주차 (6/1)~~
  - ~~초청 세미나~~
- 14주차 (6/8)
  - 18-optimization
  - 19-optimization-cache
  - 20-parallelism

- 6월 15일 목요일 오후 2시 ~

- 장소
  - 302동 105호
  - 302동 209호
  - 개인별 시험장소 배정 공지 예정 (eTL, ~6/11)
- More details will be announced through eTL

기말고사

- Q&A Session 시험 범위 가이드라인
  - 강의 내용에 대한 부연설명은 시험범위에 해당
  - 강의 내용의 advanced topic 에 대한 내용은 시험 범위에 해당되지 않음

- 예를 들면,
  - SP-session08.pdf 의 p.9~p.10 질문의 경우,
  - outside local 과 global 이 구분되어 NAT Table 에 저장되어야 하는 이유에 관한 것은 **시험 범위에 해당**
  - 이해를 돕기 위해 설명된 p.10 의 Router 에 대한 내용은 **시험 범위에 해당되지 않음**

18-optimization

Questions

Optimization 강의자료 18쪽 내용을 좀 더 자세히 설명해주시면 좋겠습니다. Value of B가 i=1일 때 왜 [3 22 16]이 되는지를 잘 모르겠습니다.

Memory Aliasing

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

Value of B:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

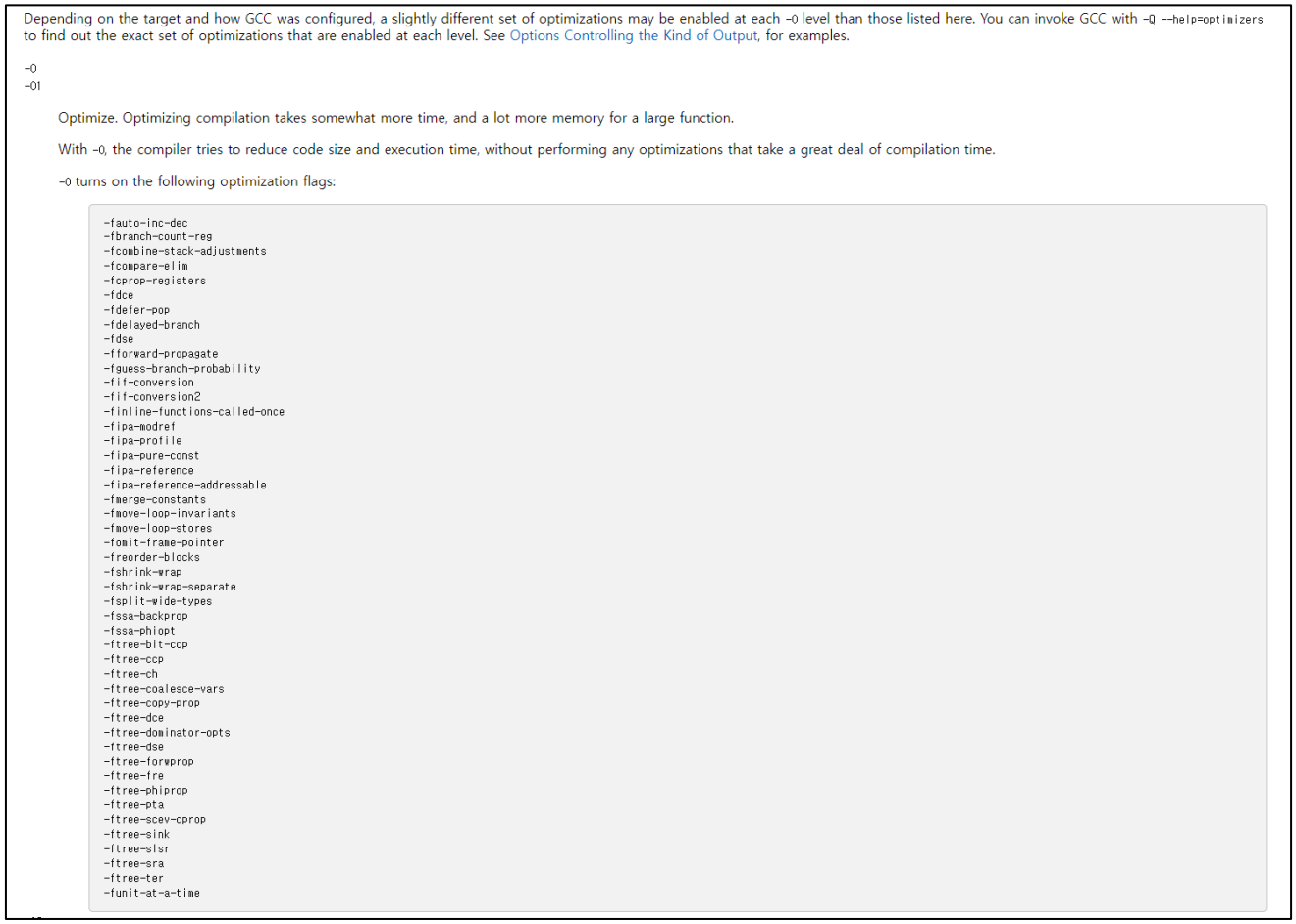
```
i = 2: [3, 22, 224]
```

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

18

Questions

[1]25p를 설명하실 때 컴파일러에서 O1 option을 주면 다음과 같이 속도가 증가하며 컴파일러가 무슨 일을 했는지, 또 추가적으로 줄일 수 있는 방법이 있는지 알아본다고 말씀하신 이후 컴파일러의 작업에 대해서는 말씀이 없으셨는데, 그냥 컴파일러는 이후에 나온 basic optimization이나 loop unrolling같은 방법이 아닌 무언가 다른 방법을 통해 최적화를 진행했다 정도로만 이해하고 넘어가면 될까요?



Depending on the target and how GCC was configured, a slightly different set of optimizations may be enabled at each -O level than those listed here. You can invoke GCC with -Q --help=optimizers to find out the exact set of optimizations that are enabled at each level. See Options Controlling the Kind of Output, for examples.

-O
-O1

Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

-O turns on the following optimization flags:

```
-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion
-fif-conversion2
-finline-functions-called-once
-fipa-modref
-fipa-profile
-fipa-pure-const
-fipa-reference
-fipa-reference-addressable
-fmerge-constants
-fmove-loop-invariants
-fmove-loop-stores
-fomit-frame-pointer
-freorder-blocks
-fshrink-wrap
-fshrink-wrap-separate
-fsplit-wide-types
-fssa-backprop
-fssa-phiopt
-ftree-bit-ccp
-ftree-ccp
-ftree-ch
-ftree-coalesce-vars
-ftree-copy-prop
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-pta
-ftree-scev-cprop
-ftree-sink
-ftree-slsr
-ftree-sra
-ftree-ter
-funit-at-a-time
```

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Questions

[2]31p를 보면 int add를 제외하고는 모두 latency bound까지 감소한 반면 int add는 그보다 두배 느린 정도까지밖에 내려가지 못했는데, 30p에 따르면 simple integer연산은 다른 연산들과 다르게 execution unit이 2개나 있음에도 불구하고 나머지 경우에 비해 느린 이유를 모르겠습니다. 정반대로 int add가 latency bound까지 빨라지고 나머지 연산들은 그 속도에 미치치 못했다고 하면 이해하도 하겠지만 31p에 나온 결과는 이해하지 못했습니다. (이는 43p에서도 나오는데, accumulator수를 늘리지 않고 unroll만 늘렸음에도 속도가 2배로 빨라지는 현상이 나타나는 이유를 모르겠습니다)

Questions

[3]또 31p에 나오는 combine4방법부터 시작해서 이후에 나오는 모든 방법들은 모두 여러개의 functional unit을 한 clock에 동시에 사용하는 것 뿐만이 아니라 pipelining까지 모두 고려한 상황에서 작성된 코드 및 실험결과라고 보면 될까요? 예를 들어 34p에서 보면 일종의 loop unrolling을 실행한 이후에 int mul을 하면 컴파일러가 똑똑하게 속도를 빠르게 해준다고 되어있는데, 애초 functional unit이 하나밖에 없기에 pipelining을 하지 않는 이상 latancy bound 밑으로 내려갈 수 없을 것 같아 질문드립니다. (이후 41p를 보면 unroll을 해봤자 execution unit에 의한 한계 이상의 최적화가 불가능하다 되어있는데, 이것을 보면 superscalar processor를 통한 최적화만 논할 뿐 pipelining을 사용하지 않은 것처럼 설명하는 것 같아 혼동이 왔습니다)

Questions

1. 18장의 unrolling에서 34쪽에서는 sequential dependency가 없어지지 않았다고 하는데, mult는 improve했습니다. 또한 35쪽에서는 FP에 대해서 발전합니다. 이런 차이의 이유가 무엇인지 모르겠습니다. Integer와 FP, Add와 Mult 간의 차이가 있는 건가요?

Questions

2. 여러 일을 수행하기 위한 개념을 다양하게 배웠는데, 정리가 잘 되지 않습니다. [A. 싱글코어->멀티코어 B. 컴퓨터 구조 pipelining (fetch, decode 등) C. functional unit을 동시에 여러 개 사용.]의 3가지를 크게 배운 것 같은데, 각각이 어떻게 개선되는지 예시와 함께 명확히 설명해주실 수 있을까요? 특히 B, C가 헷갈립니다. 예를 들면 loop unrolling의 경우 어떤 걸 개선하는 건지 잘 와닿지 않습니다.

Questions

1. 18장의 unrolling에서 34쪽에서는 sequential dependency가 없어지지 않았다고 하는데, mult는 improve했습니다. 또한 35쪽에서는 FP에 대해서 발전합니다. 이런 차이의 이유가 무엇인지 모르겠습니다. Integer와 FP, Add와 Mult 간의 차이가 있는 건가요?

5.8 Loop Unrolling

Loop unrolling is a program transformation that reduces the number of iterations for a loop by increasing the number of elements computed on each iteration. We saw an example of this with the function `psum2` (Figure 5.1), where each iteration computes two elements of the prefix sum, thereby halving the total number of iterations required. Loop unrolling can improve performance in two ways. First, it reduces the number of operations that do not contribute directly to the program result, such as loop indexing and conditional branching. Second, it exposes ways in which we can further transform the code to reduce the number of operations in the critical paths of the overall computation. In this section, we will examine simple loop unrolling, without any further transformations.

Loop Unrolling

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Latency Bound	1.0	3.0	3.0	5.0

- **Helps integer multiply**
  - below latency bound
  - Compiler does clever optimization
- **Others don't improve.** ***Why?***
  - Still sequential dependency

```
x = (x OP d[i]) OP d[i+1];
```

Loop Unrolling with Reassociation

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Compare to before

```
x = (x OP d[i]) OP d[i+1];
```

- Can this change the result of the computation?
- Yes, for FP. *Why?*

Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Unroll 2x, reassociate	2.0	1.5	1.5	3.0
Latency Bound	1.0	3.0	3.0	5.0
Throughput Bound	1.0	1.0	1.0	1.0

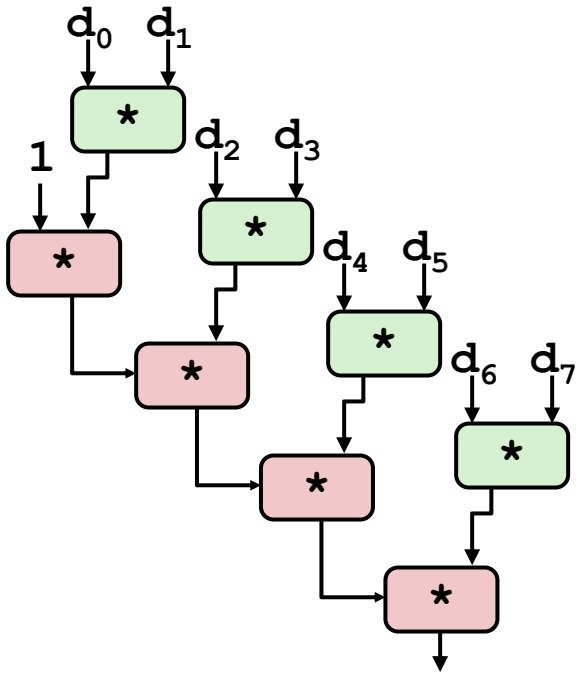
- **Nearly 2x speedup for Int *, FP +, FP ***
  - Reason: Breaks sequential dependency

```
x = x OP (d[i] OP d[i+1]);
```

  - Why is that? (next slide)

Reassociated Computation

```
x = x OP (d[i] OP d[i+1]);
```

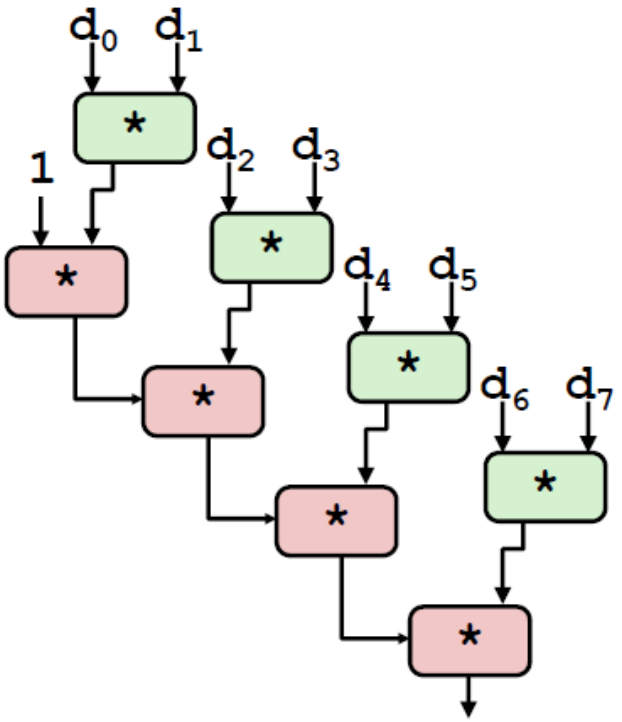


- **What changed:**
  - Ops in the next iteration can be started early (no dependency)
- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles: **CPE = D/2**
  - Measured CPE slightly worse for FP mult

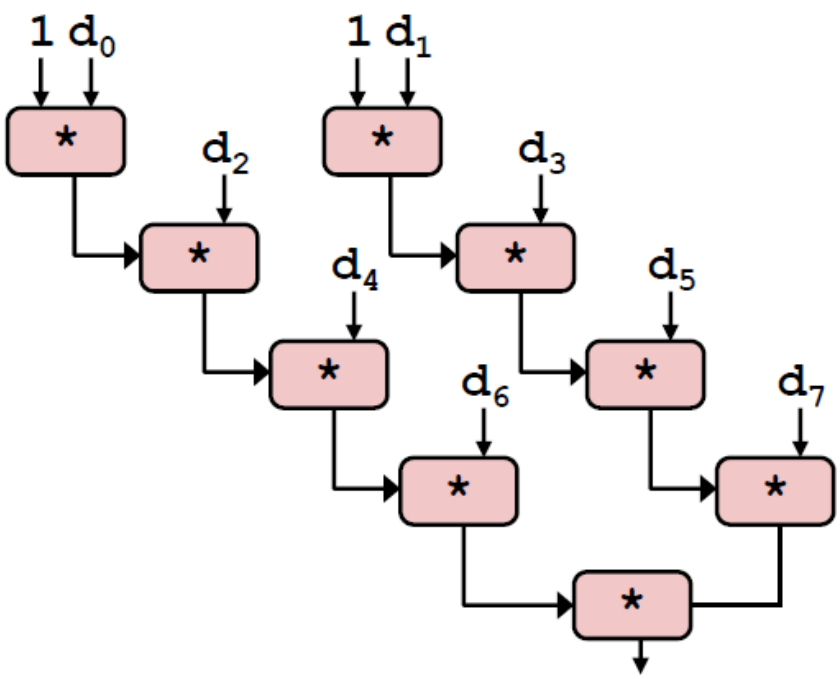
Questions

[4]37p에서 설명하는 방식처럼 unrolling을 하는 것 보다 40p처럼 unroll하는 방식이 더 좋은 이유가 stream을 나누면서 data dependency를 조금 더 확실하게 제거했다는 점으로 설명한 것 같은데, 결국 두 방법 모두 한번에 최대 두 개의 instruction을 수행할 수 있다는 점(37p 방법은 초록 1개 분홍 1개를, 40p방법은 각 stream별 분홍 node 1개씩)은 완벽히 동일함에도 결과적으로 int add와 FP mul에서 각각 25%, 17%정도씩 빨라질 수 있는 이유를 모르겠습니다.

```
x = x OP (d[i] OP d[i+1]);
```




```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



Questions

[5]위의 3번 질문과 비슷한 것 같은데, 41p에 나오는 K는 각 연산 별 functional unit의 숫자를 의미한다고 보면 되는지, 아니면 pipelining을 통한 accumulate까지 고려해 정해진 숫자인지 궁금합니다.

19-optimization-cache

20-parallelism

Questions

[1]7~8p에 나온 snoopy cache에 대한 설명이 조금 이해가 되지 않아 질문드립니다. 교수님께서는 일단 각 thread cache로 a=1, b=100이 올라간 상태에서 한 thread에서 값을 바꾸면 exclusive로 state를 변경, 다른 cache에서 값을 사용하고자 할 때 다른 thread cache에 사용하고자 하는 값이 exclusive로 표시되어있다면 그 값을 가져와서 갱신, 이후 두 위치 모두에서 state를 shared로 바꾸는 방식으로 진행된다고 설명하신 것 같았는데, invaild는 정확히 뭔지, 또 shared = readable copy이고 exclusive = writeable copy라는 내용도 이해가 가지 않아 따로 찾아보니 비슷하면서도 달라, snoopy cache에 대해 한번 정리해주셨으면 해서 질문드립니다.

Cache Coherence Problem

- Suppose two CPU cores share a physical address space
  - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

Questions

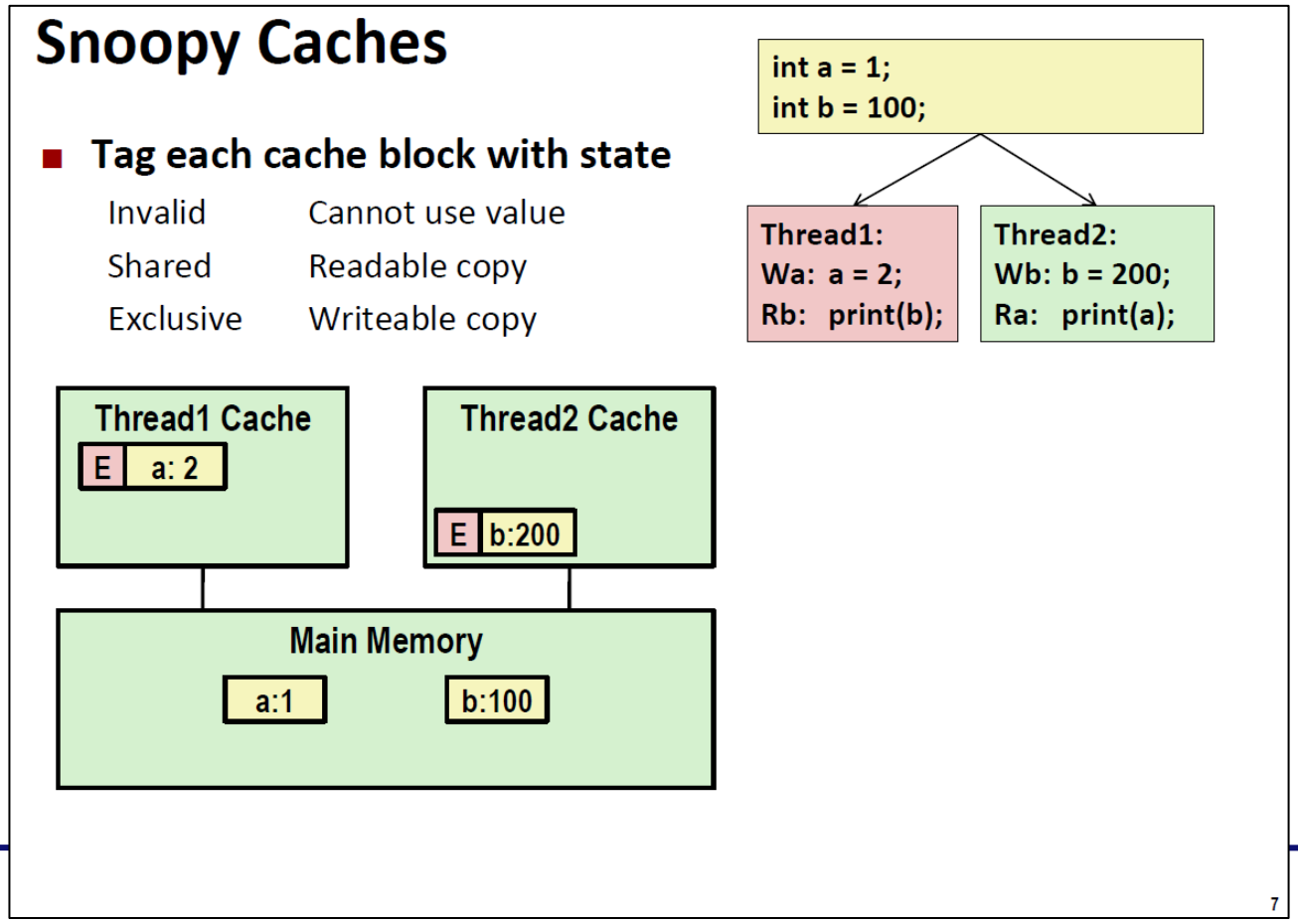
[1]7~8p에 나온 snoopy cache에 대한 설명이 조금 이해가 되지 않아 질문드립니다. 교수님께서는 일단 각 thread cache로 a=1, b=100이 올라간 상태에서 한 thread에서 값을 바꾸면 exclusive로 state를 변경, 다른 cache에서 값을 사용하고자 할 때 다른 thread cache에 사용하고자 하는 값이 exclusive로 표시되어있다면 그 값을 가져와서 갱신, 이후 두 위치 모두에서 state를 shared로 바꾸는 방식으로 진행된다고 설명하신 것 같았는데, invaild는 정확히 뭔지, 또 shared = readable copy이고 exclusive = writeable copy라는 내용도 이해가 가지 않아 따로 찾아보니 비슷하면서도 달라, snoopy cache에 대해 한번 정리해주셨으면 해서 질문드립니다.

Snoopy Protocols

- Write Invalidate Protocol:
  - Write to shared data: an invalidate is sent to all caches which snoop and *invalidate* any copies
- Write Broadcast Protocol:
  - Write to shared data: broadcast on bus, processors snoop, and *update* copies
- Write serialization: bus serializes requests
  - Bus is single point of arbitration

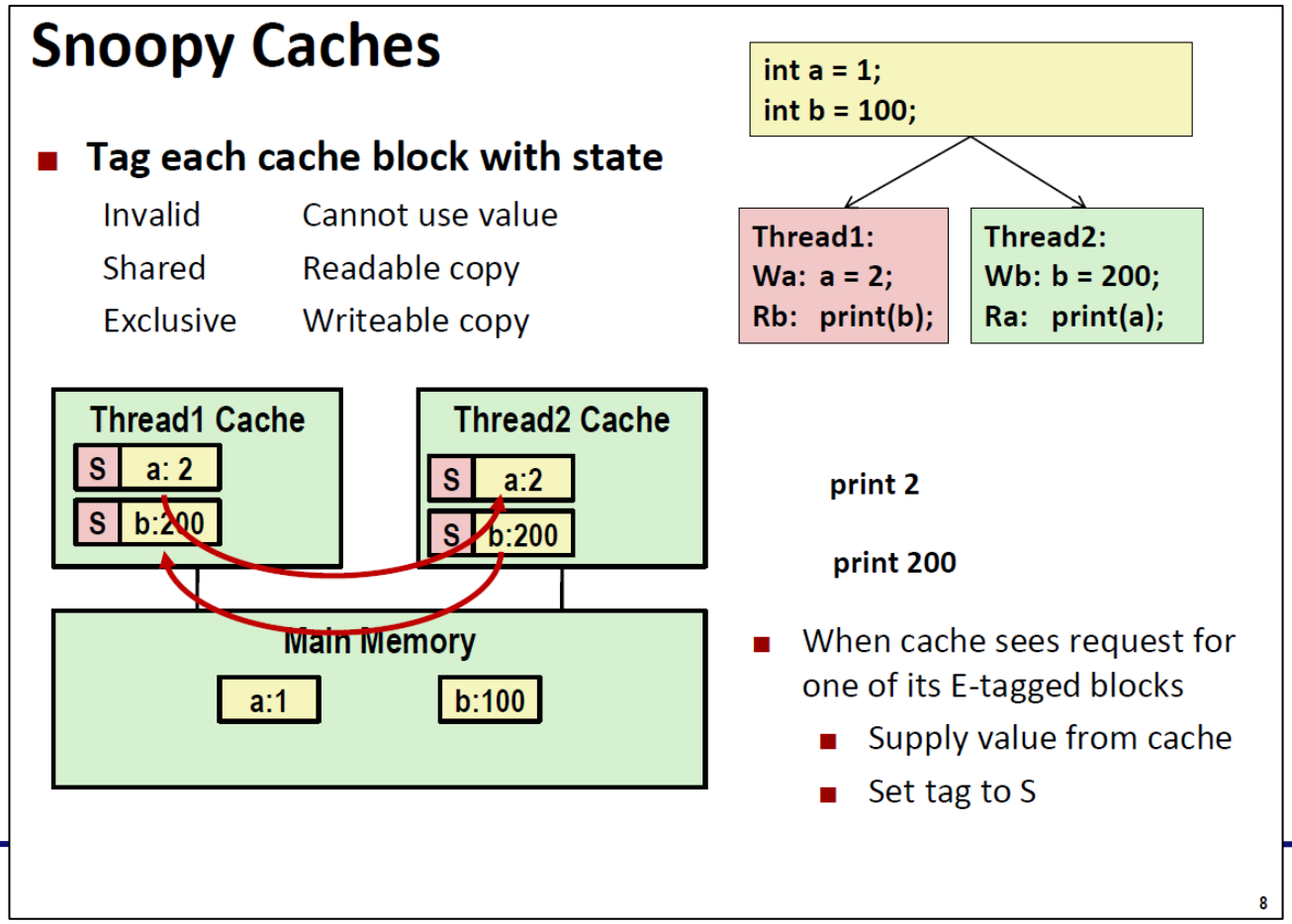
Questions

[1]7~8p에 나온 snoopy cache에 대한 설명이 조금 이해가 되지 않아 질문드립니다. 교수님께서는 일단 각 thread cache로 a=1, b=100이 올라간 상태에서 한 thread에서 값을 바꾸면 exclusive로 state를 변경, 다른 cache에서 값을 사용하고자 할 때 다른 thread cache에 사용하고자 하는 값이 exclusive로 표시되어있다면 그 값을 가져와서 갱신, 이후 두 위치 모두에서 state를 shared로 바꾸는 방식으로 진행된다고 설명하신 것 같았는데, invaild는 정확히 뭔지, 또 shared = readable copy이고 exclusive = writeable copy라는 내용도 이해가 가지 않아 따로 찾아보니 비슷하면서도 달라, snoopy cache에 대해 한번 정리해주셨으면 해서 질문드립니다.



Questions

[1]7~8p에 나온 snoopy cache에 대한 설명이 조금 이해가 되지 않아 질문드립니다. 교수님께서는 일단 각 thread cache로 a=1, b=100이 올라간 상태에서 한 thread에서 값을 바꾸면 exclusive로 state를 변경, 다른 cache에서 값을 사용하고자 할 때 다른 thread cache에 사용하고자 하는 값이 exclusive로 표시되어있다면 그 값을 가져와서 갱신, 이후 두 위치 모두에서 state를 shared로 바꾸는 방식으로 진행된다고 설명하신 것 같았는데, invaild는 정확히 뭔지, 또 shared = readable copy이고 exclusive = writeable copy라는 내용도 이해가 가지 않아 따로 찾아보니 비슷하면서도 달라, snoopy cache에 대해 한번 정리해주셨으면 해서 질문드립니다.



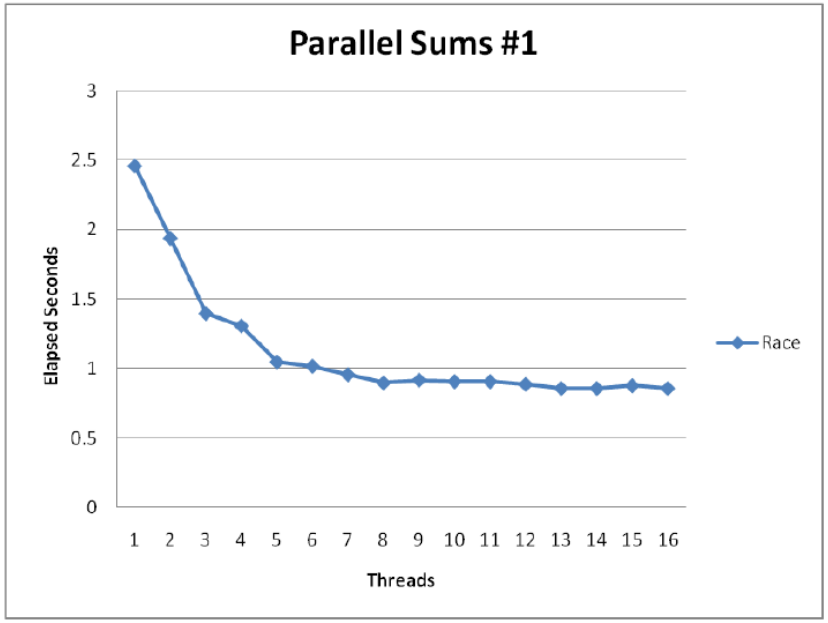
Questions

[4] 23-24p false sharing의 설명에서 spacing이 적은 경우 cache coherency를 맞추는데 걸리는 오버헤드가 있어 시간이 오래 걸린다고 이해했는데, 이 오버헤드는 앞서 나온 snoopy cache에서 각 thread cache를 순회하는데 걸리는 시간 등으로 이해하면 될까요?

Questions

[3] 16p에서 race의 경우에 thread개수에 따른 속도의 변화가 나오는데 특정 순간부터 더 빨라지지 않는 이유는 core 개수때문이라고 보면 될까요?

Unsynchronized Performance



- $N = 2^{30}$
- Best speedup = 2.86X
- Gets wrong answer when > 1 thread!

16

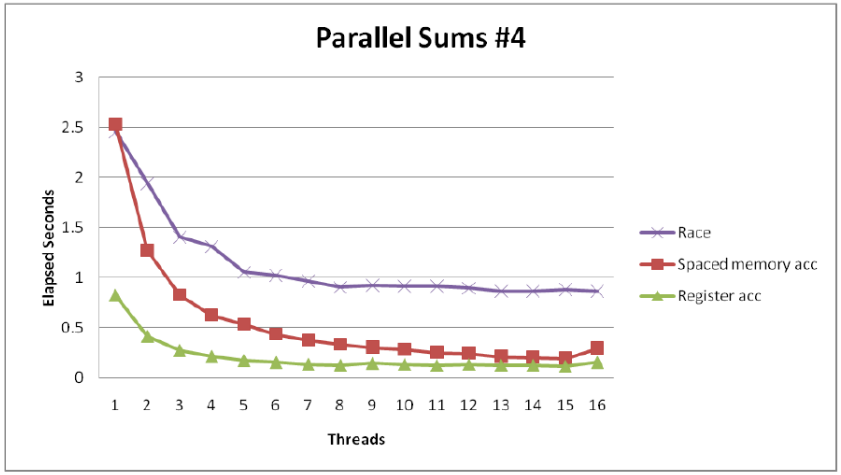
Questions

[5] 26p의 그래프를 보면 thread가 1개일 때 register에 더하는 것이 race에 더하는 것에 비해 빠르다고 나오는데 이 이유를 모르겠습니다. race의 경우 global variable 'sum'을 register에서 넣고 계속 더해주기만 하면 되는 방식이지만, register에 더해주는 경우에는 local variable 'sum'에 더해주고 굳이 이 결과를 psum에 다시 저장해주는 등 추가 작업이 필요한데 이러면 오히려 race조건이 더 빨라야 하는 것 아닌가요?
비슷한 이유로 22p에서 race랑 seperate accumulation또한 thread가 1개일때 시간이 동일한 이유 모르겠습니다. spacing이 몇이던 이 경우에는 psum[i]를 load하고 store하고 하는 추가적인 과정을 거쳐야하는데 그냥 register하나에다가 더해주기만 하는 race인 경우랑 소모 시간이 비슷한 이유가 있을까요?
사실 race는 변수 하나만 사용하지만 나머지 방식의 경우 여러개의 변수 또는 배열을 이용하는데, 쓰레드 개수와 별개로 race보다 더 빨라질 수 있는 이유를 잘 모르겠습니다.

Separate Accumulation

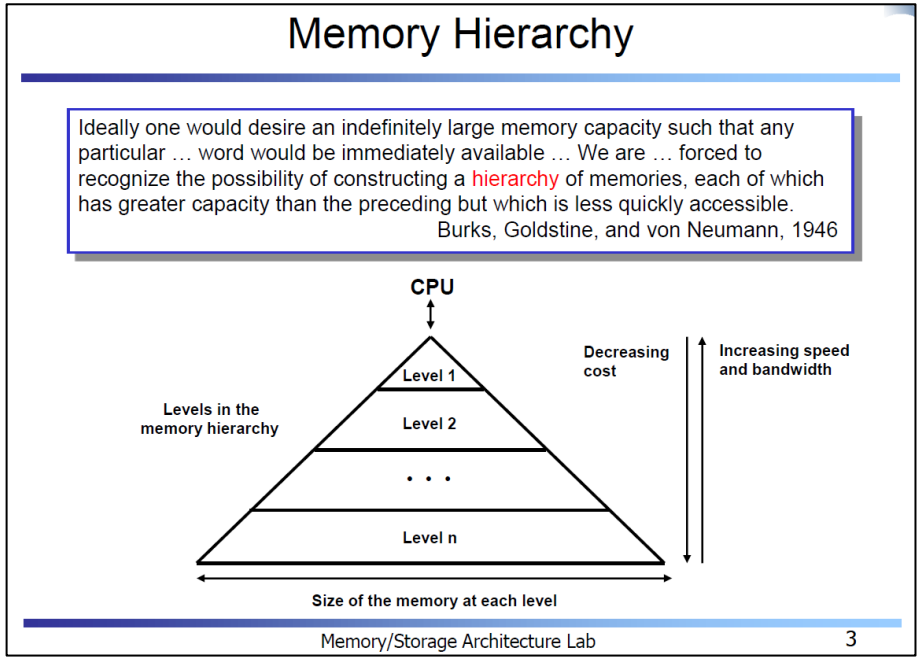
- **Method #2: Each thread accumulates into separate variable**
  - 2A: Accumulate in contiguous array elements
  - 2B: Accumulate in spaced-apart array elements
  - 2C: Accumulate in registers

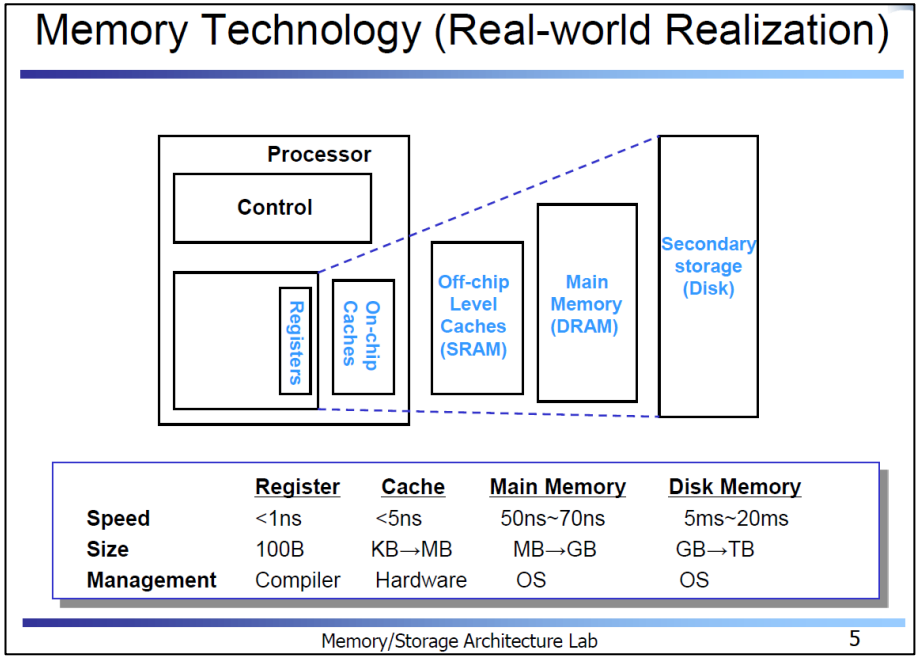
Register Accumulation Performance



Questions

[5] 26p의 그래프를 보면 thread가 1개일 때 register에 더하는 것이 race에 더하는 것에 비해 빠르다고 나오는데 이 이유를 모르겠습니다. race의 경우 global variable 'sum'을 register에서 넣고 계속 더해주기만 하면 되는 방식이지만, register에 더해주는 경우에는 local variable 'sum'에 더해주고 굳이 이 결과를 psum에 다시 저장해주는 등 추가 작업이 필요한데 이러면 오히려 race조건이 더 빨라야 하는 것 아닌가요?
비슷한 이유로 22p에서 race랑 seperate accumulation또한 thread가 1개일때 시간이 동일한 이유 모르겠습니다. spacing이 몇이던 이 경우에는 psum[i]를 load하고 store하고 하는 추가적인 과정을 거쳐야하는데 그냥 register하나에다가 더해주기만 하는 race인 경우랑 소모 시간이 비슷한 이유가 있을까요?
사실 race는 변수 하나만 사용하지만 나머지 방식의 경우 여러개의 변수 또는 배열을 이용하는데, 쓰레드 개수와 별개로 race보다 더 빨라질 수 있는 이유를 잘 모르겠습니다.





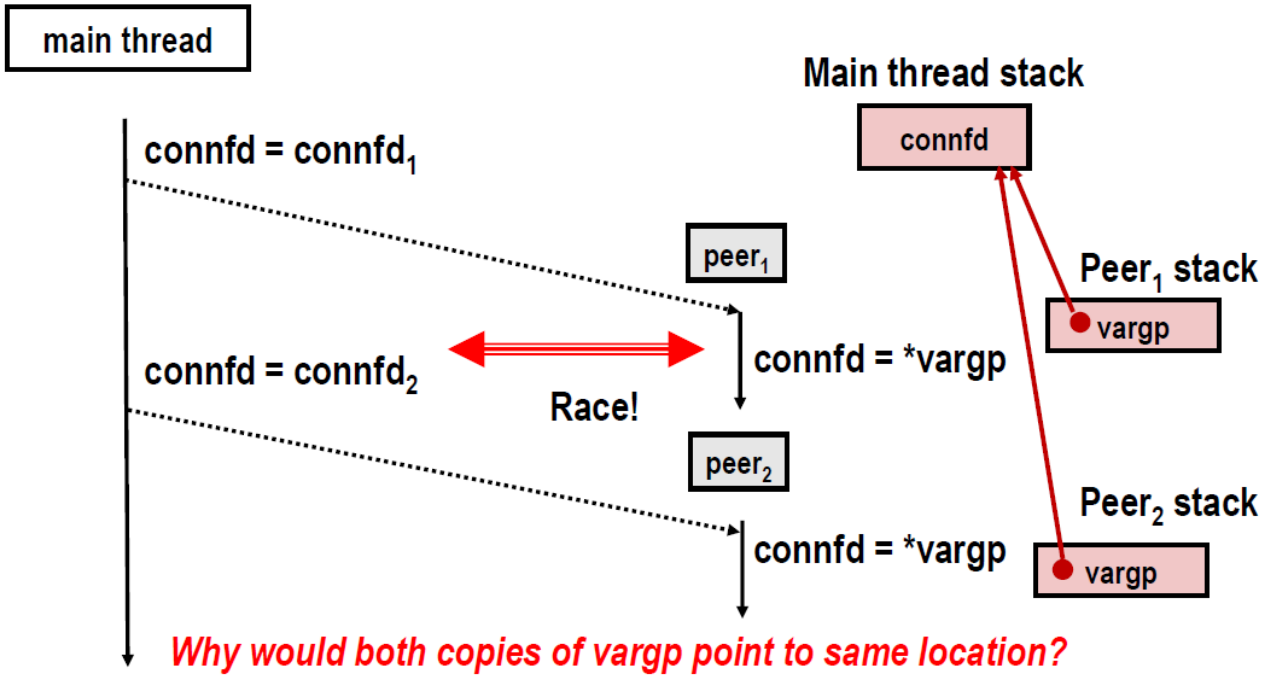
Previous chapters

Questions

2. 14-concurrent-programming.pdf 33p에서 나온 프로그램과 같이 int connfd를 반복문 안에서 선언하면 매 반복마다 초기화되기에, 만약 connfd값을 vargp에서 가져오기 전에 다음 반복으로 넘어가면 33p 아래의 설명처럼 두개의 thread가 같은 위치를 가리치는게 되는 것이 아닌, peer 1 thread에서는 scope를 나간 local 변수의 값을 가리키는 ptr를 가지게 되는 것이니 garbage value를 받게 되는것 아닌가요?

Potential Form of Unintended Sharing

```
    while (1) {
        int connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, echo_thread, (void *) &connfd);
    }
}
```



Questions

2. 14-concurrent-programming.pdf 33p에서 나온 프로그램과 같이 int connfd를 반복문 안에서 선언하면 매 반복마다 초기화되기에, 만약 connfd값을 vargp에서 가져오기 전에 다음 반복으로 넘어가면 33p 아래의 설명처럼 두개의 thread가 같은 위치를 가리치는게 되는 것이 아닌, peer 1 thread에서는 scope를 나간 local 변수의 값을 가리키는 ptr를 가지게 되는 것이니 garbage value를 받게 되는것 아닌가요?

Thread-Based Concurrent Echo Server

```
int main(int argc, char **argv) {
    int port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen=sizeof(clientaddr);
    pthread_t tid;

    int listenfd = Open_listenfd(port);
    while (1) {
       int *connfdp = Malloc(sizeof(int));
       *connfdp = Accept(listenfd,
                         (SA *) &clientaddr, &clientlen);
       Pthread_create(&tid, NULL, echo_thread, connfdp);
    }
}
```

- Spawn new thread for each client
- Pass it copy of connection file descriptor
- Note use of Malloc()!
  - Without corresponding Free()

30

Questions

16-sync-basic.pdf의 14~17p에 나오는 예시를 보면 pthread_create를 통해 i의 주소를 넘겨주고 이를 vargp라는 인자로 받은 다음 안에 있는 값(원래의 i값과 동일)을 myid에 저장하는 구조로 되어있습니다. 하지만 여기서 myid에 들어가는 0, 1이라는 값은 main thread cache에 저장되어 있는 i의 값 0, 1로부터 나온 것인데, 이러면 i또한 shared된 것으로 봐야하지 않나요? vargp라는 새로운 pointer변수를 만들었지만 그럼에도 실제 값은 main thread에 있었기에 결국 0, 1이라는 값 자체는 여러 쓰레드에서 ref했다고 보는게 맞지 않나 여쭤보고 싶습니다.

1. 16-sync-basic.pdf 16p에서 i의 주소를 넘겨주는 줄 알았는데, 알고보니 (void *)형으로 형변환한 i값을 넘겨주는 것임을 다시 공부하면서 깨달았습니다. (shared가 아님을 이해했습니다..죄송합니다)
그러면 만약 여기서 i의 값이 아닌, (void *) &i처럼 i의 주소를 넘겨주는 경우에는 shared가 맞는지 여쭤보고 싶습니다.

Questions

3. 17-sync-supplement.pdf 4~5p에 나온 writer를 favor하는 코드에 대해 교수님께서 설명하신 바에 따르면 reader가 V(&r)을 할 때 reader와 writer중 누가 들어올 지 모르는 상황이기에 이는 완벽한 writer favor방식이 아니라고 하셨는데, 그렇다면 3p에 나온 reader favor의 경우에도 writer가 끝났을 때 reader와 writer중 P(&w)를 먼저 잡은 쪽에서 우선권이 생기니 이 또한 완벽한 reader favor방식이 아니라고 생각되는데 맞을까요?
맞다면 해결방법으로 writers code에 새로운 mutex를 아래와 같이 추가하면 될꺼같은데 이 또한 맞는지 궁금합니다.

```
P(&mutex1)
P(&w);
/* Writing here */
V(&w);
V(&mutex1)
```

Questions

4. 17-sync-supplement.pdf 6p에 나온 완벽한 writer favor code를 보다 생각난건데, mutex가 적으면 적을수록 좋다는 점에서 P(&writePending); 을 추가하지 말고 아래와 같이 P(&r)과 P(&mutex1)의 순서만 바꿔줘도 해결되는 문제 아닌가요?

```
P(&mutex1);
P(&r);
readcnt = readcnt + 1;
if(readcnt == 1)
P(&w);
V(&r);
V(mutex1);
```

여러 시나리오를 생각해봤는데 deadlock같은 문제도 생기지 않는 것 같아 이렇게 해결해도 맞는지 여쭤보고 싶습니다.

Questions

17-sync 22p를 보면 thread safe의 조건이 correct result를 내는 것이라고 되어있는데, 이는 프로그래머가 생각한 목적하고 동일하면 된다는 것으로 이해해야 하는지, 아니면 다른 판단 기준이 있는지 궁금합니다. 질문이 좀 추상적인 것 같아 예시를 들면, 24p의 class2 예시를 보면 목적이 random한 값을 출력하는 것이었고 교수님 설명에 따르면 현재 24p의 코드는 여러 쓰레드에서 동일한 값이 return되는 등의 문제가 있을 수 있다고 하셨는데, 이는 next를 mutex로 감싸서 쓰레드마다 다른 값이 나오도록 만들어도 해결할 수 있는 것 아닌가요?
(인자로 넘겨줘야하는 이유가 static state라는 것 자체가 존재하면 안된다는 이유일 경우 19,20p에서 사용되는 byte_cnt또한 static state임으로 문제라고 해석해야하기에 이 이유는 아닐 것 같다고 생각했습니다.)
정리하면 24p의 코드가 class1과 같은 다른 class의 문제가 아니라 정확히 class2문제라고 판단하게 된 이유가 뭔지 궁금합니다. 오히려 인자를 넘겨주는 방법의 경우 넘겨주는 쪽에서 내부 코드를 알게되면 더이상 랜덤이 아니게 되는 반면, state를 저장하고 next를 보호하는 방법으로 갈 경우 코드를 알더라도 run time에 정해지는 state값 자체를 알 수 없기 때문에 오히려 더 랜덤에 가깝다고 볼 수도 있지 않은가요?

- *Class 1: Functions that do not protect shared variables.* We have already encountered this problem with the `thread` function in Figure 12.16, which increments an unprotected global counter variable. This class of thread-unsafe function is relatively easy to make thread-safe: protect the shared variables with synchronization operations such as P and V. An advantage is that it does not require any changes in the calling program. A disadvantage is that the synchronization operations will slow down the function.

Questions

17-sync 22p를 보면 thread safe의 조건이 correct result를 내는 것이라고 되어있는데, 이는 프로그래머가 생각한 목적하고 동일하면 된다는 것으로 이해해야 하는지, 아니면 다른 판단 기준이 있는지 궁금합니다. 질문이 좀 추상적인 것 같아 예시를 들면, 24p의 class2 예시를 보면 목적이 random한 값을 출력하는 것이었고 교수님 설명에 따르면 현재 24p의 코드는 여러 쓰레드에서 동일한 값이 return되는 등의 문제가 있을 수 있다고 하셨는데, 이는 next를 mutex로 감싸서 쓰레드마다 다른 값이 나오도록 만들어도 해결할 수 있는 것 아닌가요?
(인자로 넘겨줘야하는 이유가 static state라는 것 자체가 존재하면 안된다는 이유일 경우 19,20p에서 사용되는 byte_cnt또한 static state임으로 문제라고 해석해야하기에 이 이유는 아닐 것 같다고 생각했습니다.)
정리하면 24p의 코드가 class1과 같은 다른 class의 문제가 아니라 정확히 class2문제라고 판단하게 된 이유가 뭔지 궁금합니다. 오히려 인자를 넘겨주는 방법의 경우 넘겨주는 쪽에서 내부 코드를 알게되면 더이상 랜덤이 아니게 되는 반면, state를 저장하고 next를 보호하는 방법으로 갈 경우 코드를 알더라도 run time에 정해지는 state값 자체를 알 수 없기 때문에 오히려 더 랜덤에 가깝다고 볼 수도 있지 않은가요?

- *Class 2: Functions that keep state across multiple invocations.* A pseudo-random number generator is a simple example of this class of thread-unsafe function. Consider the pseudo-random number generator package in Figure 12.35. The `rand` function is thread-unsafe because the result of the current invocation depends on an intermediate result from the previous iteration. When we call `rand` repeatedly from a single thread after seeding it with a call to `srand`, we can expect a repeatable sequence of numbers. However, this assumption no longer holds if multiple threads are calling `rand`.

  The only way to make a function such as `rand` thread-safe is to rewrite it so that it does not use any `static` data, relying instead on the caller to pass the state information in arguments. The disadvantage is that the programmer is now forced to change the code in the calling routine as well. In a large program where there are potentially hundreds of different call sites, making such modifications could be nontrivial and prone to error.

Questions

SP-session12에서 질문드린 17-sync-advanced.pdf 26p, class 3 thread-unsafe에 대해 하나 더 여쭤보고 싶습니다. 두 방법의 차이점은 이해했는데, 결국 두 방법 다 caller에서 실제로 저장하게 될 위치를 alloc하고 free해야된다는 점은 동일하다고 보면 맞을까요? Fix2.Lock-and-copy를 보면 caller가 free는 해야된다 되어있는데 alloc에 대해서는 언급을 굳이 빼놓은 이유가 있나 해서 여쭤보고 싶습니다.

Questions

26p의 우측에 Lock-and-Copy version을 이용한 해결법 예시로 나와있는 코드를 보면 privatep라고 불리는 값을 넣어줄 주소를 보내는 것처럼 보이는데, 이는 "Fix 1. Rewrite function so caller passes address of variable to store result" 의 예시가 아닌가요? 둘의 차이를 잘 이해하지 못한 것인지, 아니면 코드 예시가 잘못된 것인지 여쭤보고 싶습니다.

Thread-Unsafe Functions (Class 3)

- **Returning a pointer to a static variable**
- **Fix 1. Rewrite function so caller passes address of variable to store result**
  - Requires changes in caller and callee
- **Fix 2. Lock-and-copy**
  - Requires simple changes in caller (and none in callee)
  - However, caller must free memory.

```
/* lock-and-copy version */
char *ctime_ts(const time_t *timep,
               char *privatep)
{
    char *sharedp;

    P(&mutex);
    sharedp = ctime(timep);
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```

Warning: Some functions like `gethostbyname` require a *deep copy*. Use reentrant `gethostbyname_r` version instead.

26

Questions

26p의 우측에 Lock-and-Copy version을 이용한 해결법 예시로 나와있는 코드를 보면 privatep라고 불리는 값을 넣어줄 주소를 보내는 것처럼 보이는데, 이는 "Fix 1. Rewrite function so caller passes address of variable to store result" 의 예시가 아닌가요? 둘의 차이를 잘 이해하지 못한 것인지, 아니면 코드 예시가 잘못된 것인지 여쭤보고 싶습니다.

```
/* lock-and-copy version */
char *ctime_ts(const time_t *timep,
               char *privatep)
{
    char *sharedp;

    P(&mutex);
    sharedp = ctime(timep);
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```

- *(textbook)*
- Some functions, such as `ctime` and `gethostbyname`, compute a result in a `static` variable and then return a pointer to that variable. If we call such functions from concurrent threads, then disaster is likely, as results being used by one thread are silently overwritten by another thread.

- Rewrite the function so that the caller passes the address of the variable in which to store the results.
  - Requires programmer to have access to the function source code.
- If the thread-unsafe function is difficult or impossible to modify, then another option is to use `lock-and-copy` technique.