UNIVERSITY OF CALIFORNIA, LOS ANGELES UCLA

BERKELEY • DAVIS • IRVINE • LOS ANGELES • RIVERSIDE • SAN DIEGO • SAN FRANCISCO SANTA BARBARA • SANTA CRUZ

CS 33
Exam #2

Before you start, make sure you have all 7 pages attached to this cover sheet (an ASCII table is on the last page if you need it).

All work and answers should be written directly on these pages, use the backs of pages if needed.

This is an open book, open notes quiz – but you cannot share books or notes.

I will follow the guidelines of the university in reporting academic misconduct – please do not cheat.

NAME: [redacted]

ID:

Problem 1: 10

Problem 2: 18

Problem 3: 10

Problem 4: 18

Problem 5: 30

Total: 86

1. ***Optimize This! (10 points):*** Consider the following C function `bar` on the left below. We would like to optimize this function as much as possible using code motion, shared subexpressions, and strength reduction. We know that `rand()` generates a random number and that `foo()` is side-effect free and deterministic. You may assume that variables `a` and `b` are not NULL and that `*b` and `a[i]` will always refer to accessible memory (i.e. no segmentation faults). Which of the following functions on the right below has the same effect as bar – and is the most optimal in terms of execution time?

```
int bar (int u, int v, int *a, int *b)   {
       int i,j,k;

       for (i=0; i<u; i++)    {
           k=rand()%4;
           for (j=0; j<v; j++)  {
               a[i]+=foo(i,k);
               *b=i*k;
           }
       }
}
```

Answer: Bar 2 (1, 2, or 3)

```
int bar1 (int u, int v, int *a, int *b)   {
       int i,j,k,x,y;

       for (i=0; i<u; i++)   {
           k=rand()%4;
           x=foo(i,k);
           y=i*k;
           for (j=0; j<v; j++)   {
               a[i]+=x;
               *b=y;
           }
       }
   }
```

```
int bar2 (int u, int v, int *a, int *b)   {
       int i,j,k,x,y;

       for (i=0; i<u; i++)   {
           k=rand()%4;
           x=foo(i,k);
           y=a[i];
           for (j=0; j<v; j++)   {
               y+=x;
           }
           a[i]=y;
       }
       *b=(u-1)*k;
   }
```

```
int bar3 (int u, int v, int *a, int *b)   {
       int i,j,k,x;

       for (i=0; i<u; i++)   {
           k=rand()%4;
           x=foo(i,k);
           *b=i*k;
           for (j=0; j<v; j++)   {
               a[i]+=x;
           }
       }
   }
```

18

2. ***Blanking Out? (20 points):*** Consider the following C code:

```
check:
| ret     |
| old ebp |  ← ebp
| edi spill |
| esi spill |
| ebx spill |
| rand y  |
| rand x  |
```

```
#include <stdlib.h>

int foo (int x, int y) { return x-y; }

int check(void) {
 int a, b, c;

 a=rand();
 b=rand();
 c=foo(rand(), rand());
 return a+b+c;
}
```

Here is the corresponding IA32 assembly code for these two functions – notice that there are some blanks in the code – you will need to fill these in to make the code work correctly. There are 6 total blanks for you to fill in. Assume that all register values before the call to *check* are needed after the call.

```
08048364 <foo>:
8048364:      55                         push   %ebp
8048365:      89 e5                      mov    %esp,%ebp
8048367:      8b 45 0c                   mov    0x8(%ebp),%eax
804836a:      0f af 45 08                sub    0xc(%ebp),%eax
804836e:      c9                         leave
804836f:      c3                         ret

08048370 <check>:
8048370:      55                         push   %ebp
8048371:      89 e5                      mov    %esp,%ebp
8048373:      83 ec 18             -2 ✗  sub    0x14 ,%esp
8048376:      89 5d f4                ✓  mov    %ebx ,0xfffffff4(%ebp)
8048379:      89 75 f8                   mov    %esi,0xfffffff8(%ebp)
804837c:      89 7d fc                   mov    %edi,0xfffffffc(%ebp)      f = -1
804837f:      83 e4 f0                   and    $0xfffffff0,%esp           e = -2
8048382:      83 ec 10                   sub    $0x10,%esp                 d = -3
8048385:      e8 26 ff ff ff             call   80482b0 <rand@plt>         c = -4
804838a:      89 c3                      mov    %eax,%ebx                  b = -5
804838c:      e8 1f ff ff ff             call   80482b0 <rand@plt>         a = -6
8048391:      89 c7                      mov    %eax,%edi                  9 = -7
8048393:      e8 18 ff ff ff             call   80482b0 <rand@plt>         8 = -8
8048398:      89 c6                      mov    %eax,%esi                  7 = -9
804839a:      e8 11 ff ff ff          ✓  call   80482b0 <rand@plt>         6 = -10
804839f:      89 74 24 04             ✓  mov    %eax ,0x4(%esp)            5 = -11
80483a3:      89 04 24                ✓  mov    %esi ,(%esp)               4 = -12
80483a6:      e8 b9 ff ff ff          ✓  call   8048364 <foo>
80483ab:      01 fb                   ✓  add    %edi,%ebx
80483ad:      01 c3                      add    %eax,%ebx
80483af:      89 d8                      mov    %ebx,%eax
80483b1:      8b 5d f4                ✓  mov    0xfffffff4(%ebp),%ebx
80483b4:      8b 75 f8                   mov    0xfffffff8(%ebp),%esi
80483b7:      8b 7d fc                   mov    0xfffffffc(%ebp),%edi
80483ba:      c9                         leave
80483bb:      c3                         ret
```

Here's the 64-bit version of the assembly code. Again, there are some blanks in the code for you to fill in – 4 total this time.

```
00000000004004a8 <foo>:
 4004a8:          89 f8                  ✓mov     %edi,%eax
 4004aa:          0f af c6               ✓sub     %esi,%eax
 4004ad:          c3                      retq

00000000004004ae <main>:
 4004ae:          48 89 5c 24 e8          mov     %rbx,0xffffffffffffffe8(%rsp)
 4004b3:          4c 89 64 24 f0          mov     %r12,0xfffffffffffffff0(%rsp)
 4004b8:          4c 89 6c 24 f8          mov     %r13,0xfffffffffffffff8(%rsp)
 4004bd:          48 83 ec 18             sub     $0x18,%rsp
 4004c1:          e8 1a ff ff ff          callq   4003e0 <rand@plt>
 4004c6:          89 c3                   mov     %eax,%ebx
 4004c8:          e8 13 ff ff ff          callq   4003e0 <rand@plt>
 4004cd:          41 89 c5                mov     %eax,%r13d
 4004d0:          e8 0b ff ff ff          callq   4003e0 <rand@plt>
 4004d5:          41 89 c4                mov     %eax,%r12d
 4004d8:          e8 03 ff ff ff          callq   4003e0 <rand@plt>
 4004dd:          89 c7                  ✓mov     %eax,%edi
 4004df:          44 89 e6                mov     %r12d,%esi
 4004e2:          e8 c1 ff ff ff          callq   4004a8 <foo>
 4004e7:          44 01 eb               ✓add     %r13d,%ebx
 4004ea:          01 c3                   add     %eax,%ebx
 4004ec:          89 d8                   mov     %ebx,%eax
 4004ee:          48 8b 1c 24             mov     (%rsp),%rbx
 4004f2:          4c 8b 64 24 08          mov     0x8(%rsp),%r12
 4004f7:          4c 8b 6c 24 10          mov     0x10(%rsp),%r13
 4004fc:          48 83 c4 18             add     $0x18,%rsp
 400500:          c3                      retq
```

3. ***A More Perfect Union (10 points)***: Consider the following **union** declaration, paying attention to memory alignment (assume the use of the Windows OS on a 64-bit machine):

```
union base {
    float num;
    char code[4];
    char *name;
  } b[2];
```

Unions allocate to biggest element!

← 8 bytes!

How much space will this array take up after the following instructions have been executed:

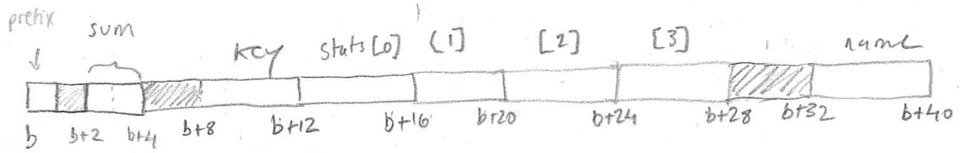
```
b[0].num=4;
b[1].num=5;
```

Space = 16 Bytes

Then - how much space will this array take up after this additional instruction has been executed:

```
b[0].code[0]='C';
```

Space = 16 Bytes



4. ***Structured Play (30 points):*** Consider the following C structure definition:

sizeOf (A) = 40 bytes = 0x28 bytes
&A = 0x 7fb ffff a10
&A[1] = 0x7fbffffa38
&A[2] = 0x7fbffffa60
&A[3] = 0x7fbffffa88

~~64×5 = 320 bytes = 0x140 Bytes~~

```
struct base {
  char prefix;  ← 1 byte
  short sum;    ← 2 bytes
  long key;     ← 8 bytes
  int stats[4]; ← 4×4 = 16 bytes
  char *name;   ← 8 bytes
} a[5];
```

This code is compiled on a 64-bit (i.e. pointers use 64 bits), little-endian architecture. For the purposes of memory alignment, treat the operating system like Windows. Using the same version of gdb that you used for your labs, we find some information (note that this is taken verbatim from the ***exact*** version of gdb on our class machines):

```
(gdb) print &a
$1 = (struct base (*)[5]) 0x7fbffffa10
                        0 1 2 3         4 5 6 7         8 9 A B         C D E F
(gdb) x/64 0x7fbffffa10
0x7fbffffa10: a[0] 0x23c6ff67   0x00000000       0x643c9869       0x00000000
0x7fbffffa20:      0x66334873   0x74b0dc51       0x19495cff       0x2ae8944a
0x7fbffffa30:      0x004006cc   0x00000000  a[1] 0x1f2900ec       0x00000000
0x7fbffffa40:      0x46e87ccd   0x00000000       0x3d1b58ba       0x507ed7ab
0x7fbffffa50:      0x2eb141f2   0x41b71efb       0x004006d3       0x00000000
0x7fbffffa60: a[2] 0xe14600e3   0x00000000       0x515f007c  a[3].sum 0x00000000
0x7fbffffa70:      0x5bd062c2   0x12200854       0x4db127f8       0x0216231b
0x7fbffffa80:      0x004006da   0x00000000  a[3] 0xcde759e8       0x00000030
0x7fbffffa90:      0x66ef438d   0x00000000       0x140e0f76       0x3352255a
0x7fbffffaa0: a[2].name 0x109cf92e 0x0ded7263    0x004006e1       0x00000000
0x7fbffffab0:      0xd79f0033   0x00000000       0x41a7c4c9       0x00000000
0x7fbffffac0:      0x6b68079a   0x4e6afb66       0x25e45d32       0x519b500d
0x7fbffffad0:      0x004006e8   0x00000000       0x00000000       0x00000000
0x7fbffffae0:      0x89e14c40 a[3].sum 0x00000030 0x00000005      0x00000004
0x7fbffffaf0:      0xbffffbe8   0x0000007f       0x00400601       0x00000001
0x7fbffffb00:      0x00000000   0x00000000       0x00000000       0x00000000

(gdb) x/32 0x4006cc
0x4006cc: 0x65687441    0x5300736e    0x74726170    0x68540061
0x4006dc: 0x65636172    0x6c654400    0x00696870    0x6273654c
0x4006ec: 0x0000736f    0x3b031b01    0x00000024    0x00000003
0x4006fc: 0xffffdb8     0x00000040    0xffffef0     0x00000080
0x40070c: 0xffffff50    0x000000a0    0x00000000    0x00000014
0x40071c: 0x00000000    0x78010001    0x08070c10    0x00000190
0x40072c: 0x00000000    0x00000024    0x0000001c    0x004004a8
0x40073c: 0x00000000    0x00000131    0x00000000    0x86100e41
```

Based on this information, fill in the correct response for these two gdb queries (note that the answer should be in decimal):

```
(gdb) print (short)a[3].sum
```

a[3].sum = 0xe859 = $\overset{e}{16^3 \times 15} + \overset{8}{16^2 \times 8} + \overset{5}{16 \times 5} + \overset{9}{9}$ = $\boxed{63577}$

```
(gdb) print a[2].name
```

a[2].name = 0x da064000

5. ***I CUDA BIN Somebody (30 points):*** Consider the CUDA code below:

block0 :
s_a= {0,1,2,4,8}
block1 :
s_a= {0,16,32,64,128}

a= {1,3,6,12,16,48,96,192}

```
#include <stdio.h>
#define BLOCK_DIMX 4
__global__ void kernel( int *a )
      {
            __shared__ int s_a[BLOCK_DIMX+1];
            int idx = blockIdx.x*blockDim.x + threadIdx.x;

            s_a[threadIdx.x+1] = a[idx];
            if (threadIdx.x == 0) {
                  s_a[0]=0;
            }
            __syncthreads();
            a[idx] = s_a[threadIdx.x+1]+s_a[threadIdx.x];
      }

int main()
      {
                                    8 ints
            int dimx = 8;
            int num_bytes = dimx*sizeof(int);
            int *d_a=0, *h_a=0; // device and host pointers
            dim3 grid, block;
            int i;

            h_a = (int*)malloc(num_bytes);
            cudaMalloc( (void**)&d_a, num_bytes );
            if( 0==h_a || 0==d_a )
            {
                  printf("couldn't allocate memory\n");
                  return 1;
            }

            h_a[0]=1;
            for (i=1; i<dimx; i++)
                  h_a[i]=h_a[i-1]*2; ← each entry is double of previous
            block.x = 4;  // 4 threads per block
            grid.x = dimx / block.x; // 2 blocks
            cudaMemcpy( d_a, h_a, num_bytes, cudaMemcpyHostToDevice );
            kernel<<<grid, block>>>( d_a );
            cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
            for(int i=0; i<dimx; i++)
                  printf("%d ", h_a[i] );
            printf("\n");
            free( h_a );
            cudaFree( d_a );
            return 0;
      }
```

h-a= {1,2,4,8,16,32,64,128}

What is the output of this code when executed on a system with a CUDA-enhanced GPU?

1 3 6 12 16 48 96 192

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL