Namn:_____

Person-nummer:_____

Systems programming and Operating systems, 2005

Test Exam

Instructions:

- Make sure that your exam is not missing any sheets, then write your name and person-nummer on the front. If you need extra pages be sure to write on those too.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 60 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is OPEN BOOK. You may use any books or notes you like. Good luck!

Problem 1. (8 points):

In this problem let `REF(x.i) --> DEF(x.k)` denote that the linker will associate an arbitrary reference to symbol x in module i to the definition of x in module k. For each example below, use this notation to indicate how the linker would resolve references the multiply defined symbol in each module. If there is a link-time error, write "ERROR". If the linker arbitrarily chooses one of the definitions, write "UNKNOWN".

A.

```
/* Module 1 */                  /* Module 2 */
int main()                      static int main=1;
{                               int p2()
}                               {
                                }
```

(a) `REF(main.1) --> DEF(_____.__)`

(b) `REF(main.2) --> DEF(_____.__)`

B.

```
/* Module 1 */                  /* Module 2 */
int x;                          double x;
void main()                     int p2()
}                               }
{                               }
```

(a) `REF(x.1) --> DEF(_____.__)`

(b) `REF(x.2) --> DEF(_____.__)`

C.

```
int x = 1;                      double x = 1.0;
void main()                     int p2()
}                               }
{                               }
```

(a) `REF(x.1) --> DEF(_____.__)`

(b) `REF(x.2) --> DEF(_____.__)`

Problem 2. (10 points):

Consider the C program below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main () {
    if (fork() == 0) {
        if (fork() == 0) {
            printf("3");
        }
        else {
            pid_t pid; int status;
            if ((pid = wait(&status)) > 0) {
                printf("4");
            }
        }
    }
    else {
        printf("2");
        exit(0);
    }
    printf("0");
    return 0;
}
```

For each of the following strings, circle whether (Y) or not (N) this string is a possible output of the program.

A. `32040` Y N

B. `34002` Y N

C. `30402` Y N

D. `23040` Y N

E. `40302` Y N

Problem 3. (16 points):

This problem tests your understanding of exceptional control flow in C programs. Assume we are running code on a Unix machine. The following problems all concern the value of the variable `counter`.

Part I (6 points)

```
int counter = 0;

int  main()
{
    int i;

    for (i = 0; i < 2; i ++){
        fork();
        counter ++;
        printf("counter = %d\n", counter);
    }

    printf("counter = %d\n", counter);
    return 0;
}
```

A. How many times would the value of `counter` be printed: _____

B. What is the value of `counter` printed in the first line? _____

C. What is the value of `counter` printed in the last line? _____

Part II (6 points)

```
pid_t pid;
int counter = 0;

void handler1(int sig)
{
    counter ++;
    printf("counter = %d\n", counter);
    fflush(stdout);   /* Flushes the printed string to stdout */
    kill(pid, SIGUSR1);
}

void handler2(int sig)
{
    counter += 3;
    printf("counter = %d\n", counter);
    exit(0);
}

main() {
    signal(SIGUSR1, handler1);
    if ((pid = fork()) == 0) {
        signal(SIGUSR1, handler2);
        kill(getppid(), SIGUSR1);
        while(1) {};
    }
    else {
        pid_t p; int status;
        if ((p = wait(&status)) > 0) {
            counter += 2;
            printf("counter = %d\n", counter);
        }
    }
}
```

What is the output of this program?

Part III (4 points)

```
int counter = 0;

void handler(int sig)
{
    counter ++;
}


int  main()
{
    int i;

    signal(SIGCHLD, handler);

    for (i = 0; i < 5; i ++){
        if (fork() == 0){
            exit(0);
        }
    }

    /* wait for all children to die */
    while (wait(NULL) != -1);

    printf("counter = %d\n", counter);
    return 0;
}
```

A. Does the program output the same value of `counter` every time we run it? `Yes` `No`

B. If the answer to A is `Yes`, indicate the value of the `counter` variable. Otherwise, list all possible values of the `counter` variable.

Answer: `counter` = _____

Problem 4. (4 points):

Consider the following C program. (For space reasons, we are not checking error return codes. You can assume that all functions return normally.)

```
int val = 10;

void handler(sig)
{
    val += 5;
    return;
}

int main()
{
    int pid;

    signal(SIGCHLD, handler);
    if ((pid = fork()) == 0) {
        val -= 3;
        exit(0);
    }
    waitpid(pid, NULL, 0);
    printf("val = %d\n", val);
    exit(0);
}
```

What is the output of this program? `val = _____`

Problem 5. (10 points):

Consider an allocator that uses an implicit free list. Each memory block, either allocated or free, has a size that is a multiple of eight bytes. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer and is represented in units of bytes. The usage of the remaining 3 lower order bits is as follows:

- `bit 0` indicates the use of the current block: 1 for allocated, 0 for free.
- `bit 1` indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- `bit 2` is unused and is always set to be 0.

Five helper routines are defined to facilitate the implementation of `free(void *p)`. The functionality of each routine is explained in the comment above the function definition. Fill in the body of the helper routines the code section label that implement the corresponding functionality correctly.

```
/* given a pointer p to an allocated block, i.e., p is a
   pointer returned by some previous malloc()/realloc() call;
   returns the pointer to the header of the block*/
void * header(void* p)
{
  void *ptr;

  _______;
  return ptr;
}

A. ptr=p-1
B. ptr=(void *)((int *)p-1)
C. ptr=(void *)((int *)p-4)


/* given a pointer to a valid block header or footer,
   returns the size of the block */
int size(void *hp)
{
  int result;

  _______;
  return result;
}

A. result=(*hp)&(~7)
B. result=((*(char *)hp)&(~5))<<2
C. result=(*(int *)hp)&(~7)
```

```
/* given a pointer p to an allocated block,i.e. p is
   a pointer returned by some previous malloc()/realloc() call;
   returns the pointer to the footer of the block*/
void * footer(void *p)
{
  void *ptr;

  _______;
  return ptr;
}
```

A. ptr=p+size(header(p))-8
B. ptr=p+size(header(p))-4
C. ptr=(int *)p+size(header(p))-2

```
/* given a pointer to a valid block header or footer,
   returns the usage of the currect block,
   1 for allocated, 0 for free */
int allocated(void *hp)
{
  int result;

  ______;
  return result;
}
```

A. result=(*(int *)hp)&1
B. result=(*(int *hp)&0
C. result=(*(int *)hp)|1

```
/* given a pointer to a valid block header,
   returns the pointer to the header of previous block in memory */
void * prev(void *hp)
{
  void *ptr;

  ______;
  return ptr;
}
```

A. ptr = hp - size(hp)
B. ptr = hp - size(hp-4)
C. ptr = hp - size(hp-4) + 4

Problem 6. (12 points):

The following problem concerns the way virtual addresses are translated into physical addresses.

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Virtual addresses are 16 bits wide.
- Physical addresses are 13 bits wide.
- The page size is 512 bytes.
- The TLB is 8-way set associative with 16 total entries.
- The cache is 2-way set associative, with a 4 byte line size and 16 total lines.

In the following tables, **all numbers are given in hexadecimal**. The contents of the TLB, the page table for the first 32 pages, and the cache are as follows:

TLB			
Index	Tag	PPN	Valid
0	09	4	1
	12	2	1
	10	0	1
	08	5	1
	05	7	1
	13	1	0
	10	3	0
	18	3	0
1	04	1	0
	0C	1	0
	12	0	0
	08	1	0
	06	7	0
	03	1	0
	07	5	0
	02	2	0

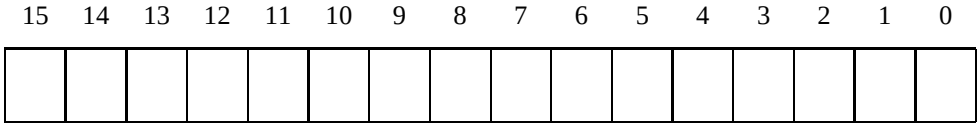
Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	6	1	10	0	1
01	5	0	11	5	0
02	3	1	12	2	1
03	4	1	13	4	0
04	2	0	14	6	0
05	7	1	15	2	0
06	1	0	16	4	0
07	3	0	17	6	0
08	5	1	18	1	1
09	4	0	19	2	0
0A	3	0	1A	5	0
0B	2	0	1B	7	0
0C	5	0	1C	6	0
0D	6	0	1D	2	0
0E	1	1	1E	3	0
0F	0	0	1F	1	0

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	19	1	99	11	23	11	00	0	99	11	23	11
1	15	0	4F	22	EC	11	2F	1	55	59	0B	41
2	1B	1	00	02	04	08	0B	1	01	03	05	07
3	06	0	84	06	B2	9C	12	0	84	06	B2	9C
4	07	0	43	6D	8F	09	05	0	43	6D	8F	09
5	0D	1	36	32	00	78	1E	1	A1	B2	C4	DE
6	11	0	A2	37	68	31	00	1	BB	77	33	00
7	16	1	11	C2	11	33	1E	1	00	C0	0F	00

Part 1

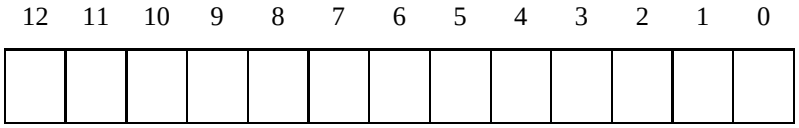
A. The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

VPO	The virtual page offset
VPN	The virtual page number
TLBI	The TLB index
TLBT	The TLB tag



B. The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

PPO	The physical page offset
PPN	The physical page number
CO	The block offset within the cache line
CI	The cache index
CT	The cache tag



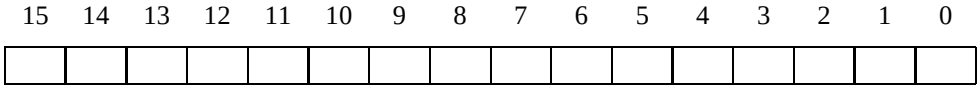
Part 2

For the given virtual address, indicate the TLB entry accessed, the physical address, and the cache byte value returned **in hex**. Indicate whether the TLB misses, whether a page fault occurs, and whether a cache miss occurs.

If there is a cache miss, enter "-" for "Cache Byte returned". If there is a page fault, enter "-" for "PPN" and leave parts C and D blank.

Virtual address: `1DDE`

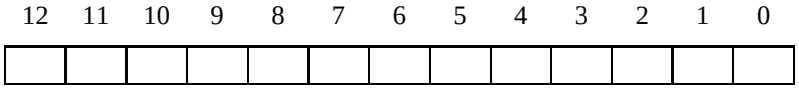
A. Virtual address format (one bit per box)



B. Address translation

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

C. Physical address format (one bit per box)



D. Physical memory reference

Parameter	Value
Byte offset	0x
Cache Index	0x
Cache Tag	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x