CSPB 2400 - Park - Computer Systems

Dashboard / My courses / 2241:CSPB 2400 / 6 May - 12 May / Exam #4: Finals

Started on	Monday, 6 May 2024, 11:58 AM
State	Finished
Completed on	Monday, 6 May 2024, 12:36 PM
Time taken	38 mins 36 secs
Marks	80.83/100.00
Grade	**8.08** out of 10.00 (**81**%)

Question **1**

Correct

Mark 10.00 out of 10.00

How many “hello” output lines does this program print?

```
#include "csapp.h"

void doit()
{
  if (Fork() == 0) {
    Fork();
    printf("hello\n");
    return;
  }
  return;
}

int main()
{
  doit();
  printf("hello\n");
  exit(0);
}
```

Answer: 5 ✓

Correct - The program consists of three processes: the original parent, its child, and its grandchild. The parent executes a single **printf**. However, because of the **return** statement, the child and grandchild each execute two **printf** statements. Thus, the program prints a total of five output lines.

The program consists of three processes: the original parent, its child, and its grandchild. The parent executes a single **printf**. However, because of the **return** statement, the child and grandchild each execute two **printf** statements. Thus, the program prints a total of five output lines.

The correct answer is: 5

Question **2**

Correct

Mark 7.00 out of 7.00

Given the following program:

```
#include "csapp.h"


int foo(int n)
{
  int i;
  for (i = 0; i < n; i++) {
     fork()
  }
  printf("hello\n");
  exit(0);
}
```

How many lines of output when n == 4?

16

Your last answer was interpreted as follows: 16

Correct answer, well done.
The code above will create 2^N processes, or 16 in this case.

A correct answer is 16, which can be typed in as follows: 16

Question **3**

Correct

Mark 10.00 out of 10.00

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

```
main() {
    pid_t pid; int status;
    printf("1");
    if (fork() == 0) {
        if (fork() == 0) {
           printf("2");
        }
    }
    else {
         printf("3");
         if (fork() == 0) {
            printf("4");
            exit(0);
         }
         else {
            wait(&status);
            printf("5");
         }
    }
    printf("6");
    return 0;
}
```

Answer the following questions. Note that incorrect answers count against you, so don't guess. You won't be penalized for selecting a blank answer, but you won't get points either.

a) Can 6 be printed before 5? yes ✓

b) Will 5 always be printed after 4? no ✓

c) Can 6 be printed before 2? yes ✓

d) Will 5 always be printed after 3? yes ✓

e) Can 5 be printed before 2? yes ✓

Question **4**

Incorrect

Mark 0.00 out of 10.00

Consider the following C program. For space reasons, we are not checking error return codes, but you can assume that the code executes correctly.

```
pid_t pid;

void bar(int sig) {
  fprintf(stderr,"poke");
  exit(0);
}

void baz(int sig) {
  fprintf(stderr,"is just");
  exit(0);
}

void foo(int sig) {
  fprintf(stderr,"sushi");
  fflush(stdout);   /* Flushes the printed string to stdout */
  kill(pid, SIGUSR1);
}

main() {
  signal(SIGUSR1, foo);
  signal(SIGCHLD, baz);
  pid = fork();
  sleep(1);
  if (pid == 0) {
    signal(SIGUSR1, bar);
    kill(getppid(), SIGUSR1);
    while(1) {};
  }
  else {
    pid_t p; int status;
    if ((p = wait(&status)) < 0) {
      fprintf(stderr,"in a bowl");
    }
  }
}
```

Drag and drop the words below to provide the output generated by this program

poke ✖ sushi ✖ in a bowl ✖

poke | is just | sushi | in a bowl

Your answer is incorrect.

The correct answer is:
Consider the following C program. For space reasons, we are not checking error return codes, but you can assume that the code executes correctly.

```
pid_t pid;

void bar(int sig) {
  fprintf(stderr,"poke");
  exit(0);
}

void baz(int sig) {
  fprintf(stderr,"is just");
  exit(0);
}

void foo(int sig) {
  fprintf(stderr,"sushi");
  fflush(stdout);   /* Flushes the printed string to stdout */
  kill(pid, SIGUSR1);
}

main() {
  signal(SIGUSR1, foo);
  signal(SIGCHLD, baz);
  pid = fork();
  sleep(1);
  if (pid == 0) {
    signal(SIGUSR1, bar);
    kill(getppid(), SIGUSR1);
    while(1) {};
  }
  else {
    pid_t p; int status;
    if ((p = wait(&status)) < 0) {
      fprintf(stderr,"in a bowl");
    }
  }
}
```

Drag and drop the words below to provide the output generated by this program

[sushi] [poke] [is just]

Question **5**

Partially correct

Mark 2.50 out of 5.00

Organize the steps taken by the MMU when loading data from memory to a register into the proper order.

- The CPU generates a virtual address ✓
- The MMU fetches the appropriate PTE from the TLB ✓
- The data is retrieved from memory ✗
- The MMU translates the virtual address to physical address ✗

Not all choices below will be used.

The CPU generates a physical address | The data is retrieved from the TLB

Your answer is partially correct.

You have correctly selected 2.
The correct answer is:
Organize the steps taken by the MMU when loading data from memory to a register into the proper order.

[The CPU generates a virtual address]

[The MMU fetches the appropriate PTE from the TLB]

[The MMU translates the virtual address to physical address]

[The data is retrieved from memory]

Not all choices below will be used.

Question **6**

Partially correct

Mark 3.33 out of 5.00

What actions occur when an **exec()** system call is performed? You will be penalized for incorrect answers.

Select one or more:

- [✓ a. the **.text** and **.data** sections are mapped to the executable file] ✓
- [✓ b. new area structs are created for the code, data and so on] ✓
- c. the program code is copied into memory
- d. static libraries are copied into memory
- [✓ e. the program counter is set to the starting address] ✓
- [✓ f. a signal is sent to the parent process] ✗
- [✓ g. old area structs of existing process are removed] ✓

Your answer is partially correct.

You have selected too many options.
See section 9.8.3 of the text.

The correct answers are: old area structs of existing process are removed, new area structs are created for the code, data and so on, the **.text** and **.data** sections are mapped to the executable file, the program counter is set to the starting address

Question **7**

Correct

Mark 3.00 out of 3.00

The best-fit method chooses the largest free block into which the requested segment fits.

Select one:

- True
- [✓ False] ✓

The best fit method chooses the block most close to the needed size.

The correct answer is 'False'.

Question **8**

Correct

Mark 3.00 out of 3.00

Using the first-fit algorithm on a free list that is ordered according to increasing block sizes is equivalent to using the best-fit algorithm.

Select one:

[✓ True] ✓

False

True - in this method, the first block that fits will also be the best block that fits.

The correct answer is 'True'.

Question **9**

Correct

Mark 2.00 out of 2.00

In an allocator using an explicit free list, the header for a block contains a bit indicating the block is allocated or free and

a pointer to the next block

[✓ the size of the current block] ✓

the size of the next block

Mark 1.00 out of 1.00

The correct answer is: the size of the current block

.

Question **10**

Correct

Mark 10.00 out of 10.00

The following question concerns the **implicit** memory allocator discussed in the lectures and textbook and your lab and tests your knowledge of the implicit list data structure and boundary tag method used in that memory allocator.

The output below shows the state of the heap of the implicit allocator described in the text, lectures and implemented in your lab.

```
Heap (0x100437008):
0x100437008: header: [8:a] footer: [8:a]
0x100437010: header: [56:a] footer: [56:a]
0x100437048: header: [64:a] footer: [64:a]
0x100437088: header: [64:a] footer: [64:a]
0x1004370c8: header: [24:a] footer: [24:a]
0x1004370e0: header: [3888:f] footer: [3888:f]
0x100438010: EOL
```

This data shows the state of the implicit allocator heap after executing the following code:

```
char *p1 = malloc( 48 ✓ );

char *p2 = malloc( 56 ✓ );

char *p3 = malloc( 56 ✓ );


char *p4 = malloc( 16 ✓ );
```

Fill in the numeric values with the **largest possible value** that would result in the heap output shown above assuming that the malloc implementation

Question **11**

Partially correct

Mark 5.00 out of 10.00

The following question concerns the **implicit** memory allocator discussed in the lectures and textbook and your lab and tests your knowledge of the implicit list data structure and boundary tag method used in that memory allocator.

The output below shows the state of the heap of the implicit allocator described in the text, lectures and implemented in your lab.

```
Heap (0x10818d008):
0x10818d008: header: [8:a] footer: [8:a]
0x10818d010: header: [56:a] footer: [56:a]
0x10818d048: header: [24:a] footer: [24:a]
0x10818d060: header: [40:f] footer: [40:f]
0x10818d088: header: [24:a] footer: [24:a]
0x10818d0a0: header: [3952:f] footer: [3952:f]
0x10818e010: EOL
```

This data shows the state of the implicit allocator heap after executing the following code:

```
char *p1 = malloc( 48 ✓ );
char *p2 = malloc( 16 ✗ );
char *p3 = malloc( 32 ✗ );
free(p2);
char *p4 = malloc( 16 ✓ );
```

Fill in the numeric values with the **largest possible value** that would result in the heap output shown above assuming that the malloc implementation

Question **12**

Correct

Mark 10.00 out of 10.00

Assume the following:

- The memory is byte addressable.
- Memory accesses are to 1-byte words (not to 4-byte words).
- Virtual addresses are 17 bits wide.
- Physical addresses are 16 bits wide.
- The page size is 4096 bytes.
- The TLB is 4-way associative tlb (E=4) with 4 sets (S=4) and a total of 16 entries .
- The TLB and a portion of the page table contents are as shown below

TLB

Set #	Way #0	Way #1	Way #2	Way #3
0:	V=Y;Tag=0x4 PPN=0x1	V=Y;Tag=0x5 PPN=0x8	V=Y;Tag=0x6 PPN=0x9	--
1:	V=Y;Tag=0x1 PPN=0xf	--	--	--
2:	V=Y;Tag=0x2 PPN=0x3	V=Y;Tag=0x3 PPN=0xe	V=Y;Tag=0x5 PPN=0xb	V=Y;Tag=0x7 PPN=0xd
3:	V=Y;Tag=0x3 PPN=0xa	V=Y;Tag=0x4 PPN=0x2	V=Y;Tag=0x5 PPN=0x0	V=Y;Tag=0x7 PPN=0x7

Page Table

VPN	PPN	V?	VPN	PPN	V?
0x02	0x6	Y	0x10	0x1	Y
0x05	0xf	Y	0x13	0x2	Y
0x06	0x4	Y	0x14	0x8	Y
0x07	0xc	Y	0x16	0xb	Y
0x0a	0x3	Y	0x17	0x0	Y
0x0b	0x5	Y	0x18	0x9	Y
0x0e	0xe	Y	0x1e	0xd	Y
0x0f	0xa	Y	0x1f	0x7	Y

Assume that memory address **0x1f7** has been referenced by a load instruction. Determine the virtual page number (VPN) and use that to compute the TLB index and tag that would be used to check the TLB for the translation entry. Indicate if the entry is in the TLB (Y/N).

Indicate if the memory reference has a valid entry in the page table whether it hits in the TLB or not.

Use the information from the page table to translate the VPN to a physical page number (PPN) and then the valid physical address (PA).

For entries that can not be determined (*e.g.* the PPN or PA if a translation doesn't exist), enter "-".

Virtual Page Number (VPN)	0x 00 ✓
Virtual Page Offset (VPO)	0x 1f7 ✓
TLB Index (TLBI)	0x 0 ✓
TLB Tag (TLBT)	0x 0 ✓
TLB Hit (Y/N)?	no ✓
Valid Entry in Page Table (Y/N)?	no ✓
Physical Page Number (PPN)	0x - ✓
Physical Address (PA)	0x - ✓

Question **13**

Correct

Mark 5.00 out of 5.00

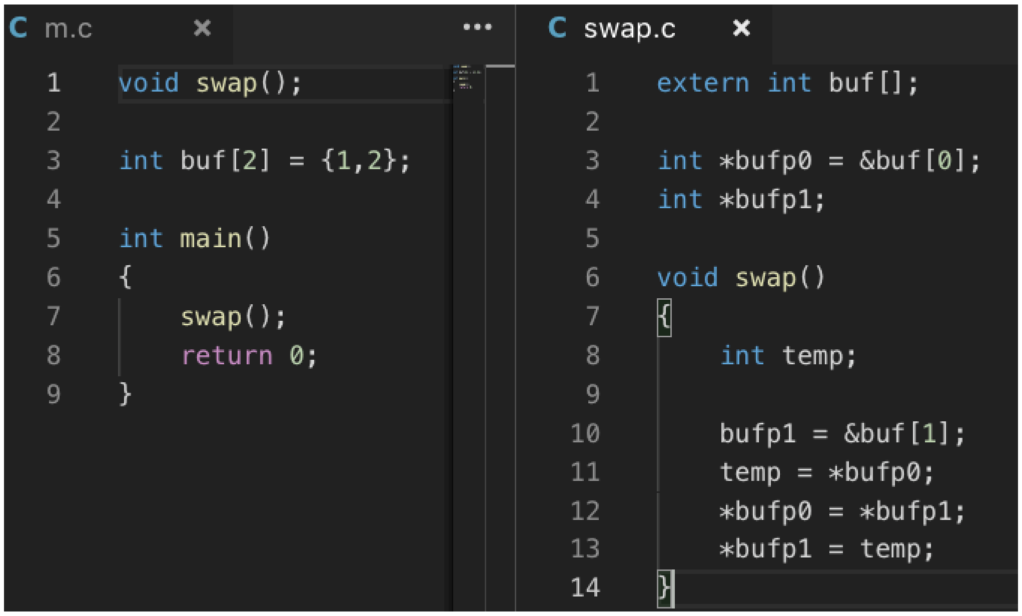
When linking multiple files containing duplicate symbols and given a strong ✓ symbol and multiple weak ✓ symbols with the same name, choose the strong ✓ one.

Question **14**

Correct

Mark 10.00 out of 10.00

Consider the following two C files (m.c and swap.c)



m.c

```c
void swap();

int buf[2] = {1,2};

int main()
{
    swap();
    return 0;
}
```

swap.c

```c
extern int buf[];

int *bufp0 = &buf[0];
int *bufp1;

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

For each symbol referenced in **swap.o**, specify whether it will or will not have a **.symtab** entry. If so, indicate the module that defines the symbol, the symbol type and the section. You are penalized for incorrect answers, so don't just guess. If there is no valid answer for a question, you can select "**No valid answer**".

Symbol	.symtab entry?	Symbol type	Module where defined	Section
buf	YES ✓	extern ✓	m.o ✓	.data ✓
bufp0	YES ✓	global ✓	swap.o ✓	.data ✓
bufp1	YES ✓	global ✓	swap.o ✓	COMMON ✓
swap	YES ✓	global ✓	swap.o ✓	.text ✓
temp	NO ✓	No valid answer ✓	No valid answer ✓	No valid answer ✓

Local symbols, like **temp** are not defined in the symbol table.