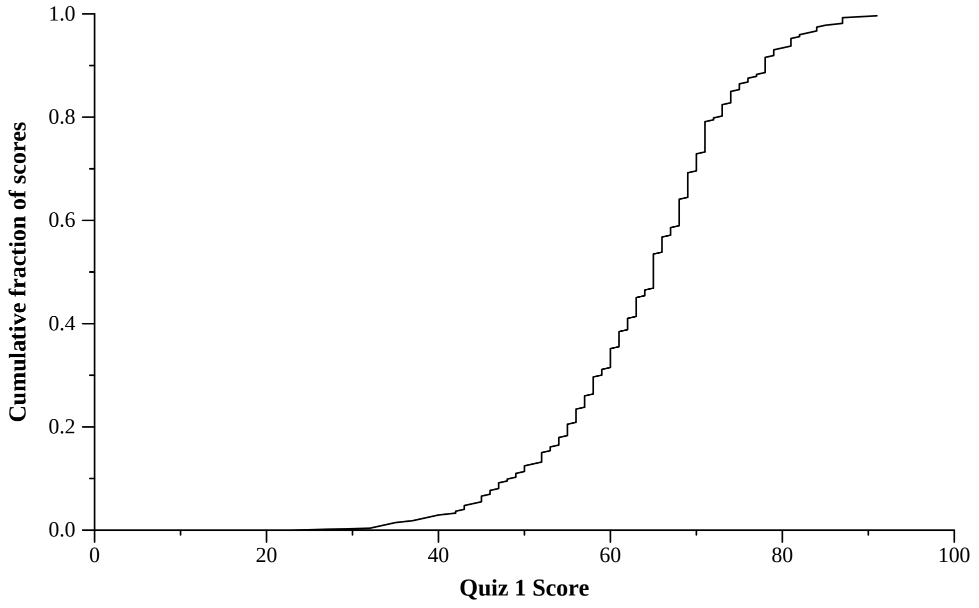
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2004

Quiz I



1-3 (xx/30)	4-5 (xx/20)	6-10 (xx/50)	Total (xx/100)

Name: Solutions

I Reading questions

1. [10 points]: Which of the following is an example of a primitive designed for sequence coordination in UNIX (reading #6, "The UNIX time-sharing system")?

(Circle ALL that apply)

[✓ **A.**] wait

[✓ **B.**] read on a pipe

C. lseek

D. mount

Using wait a parent can wait on the completion of a child, allowing the parent to coordinate its sequence with respect to the child. Using read on a pipe a process can wait for input from another process, coordinating the sequencing between the two ends of the pipe. Mount and lseek may use sequence coordination in their internal implementation, but they aren't designed for sequence coordination: lseek sets the file pointer to a specified offset in the given file, and mount hooks the specified file system into the existing file system name space.

2. [12 points]: The Flash paper ("Flash: An efficient and portable Web server," reading #7), describes the architectures: SPED, AMPED, MP, and MT. For each of the following statements, circle the architectures that apply to the statement:

(Circle ALL that apply)

A. Suppose a single client issues requests serially (i.e., starting the next request after the response on the current request has been received) for files located on the disk. Which of the following architectures provides a significant improvement in throughput over the SPED architecture.
Choose from: MP, MT, AMPED, None, All.
None. *With requests that don't overlap in time, there aren't any opportunities for any of the architectures to overlap their work, and since all requests are for files that are on the disk, it matters little whether or not there is a shared cache, so the correct answer is None.*

B. Provides the best performance when the cache miss rate is high when serving multiple clients.
Choose from: SPED, MP.
MP. *When the cache miss rate is high there will be a lot of disk waits, so the MP architecture will get a lot of chances to do those waits in parallel and its lack of a shared cache won't slow it down much. The SPED architecture, on the other hand, must do its disk waits sequentially and its single cache won't help much, so it operates at a significant disadvantage.*

C. Makes use multiple times of the UNIX fork supervisor call.
Choose from: SPED, APMED, MT, MP.
AMPED and MP. *SPED and MT each use just one process, whereas MP and AMPED both use several. So AMPED and MP are the ones that call the UNIX fork() entry multiple times.*

D. Assume that the delay observed for a client for a web request for a cached page is broken down as follows: 100 ms network latency to deliver the request packet to the server, 1ms of processing time at the server, 100 ms of network latency to deliver the page to the client in a return packet.

A number of clients issue requests at the same time. Which of the following architectures can process more than one request every 201ms.
Choose from: MP, MT, SPED, AMPED, None, All.
All. *Despite the high network latency, several requests from different clients may arrive at the server at about the same time. Since all requests are for cached pages, all the server architectures are processor-limited so every architecture can handle up to 1 request every millisecond.*

3. [8 points]: If eight stations on an Ethernet ("Ethernet: distributed packet switching for local computer networks", reading #8) all want to transmit one packet, which of the following statements is true?

(Circle best answer)

A. It is guaranteed that all transmissions will succeed.

[✓ **B.**] It is most likely (i.e., with high probability) that all stations will eventually end up being able to transmit their data successfully.

C. Some of the transmissions may eventually succeed, but it is likely some may not.

D. It is likely that none of the transmissions will eventually succeed.

There may be repeated collisions, or the power may fail, or a plumber may sever the cable. In the real world there is never a guarantee of success, so A must be wrong. On the other hand, the random exponential back-off algorithm used in the Ethernet makes it very likely that all the stations will eventually get their packets through, so B should be circled, rather than C or D.

II Local remote procedure call

Ben Bitdiddle is hired to enforce modularity in a large banking application. He splits the program into two pieces: the client and the server. Ben wants to use remote procedure calls to communicate between the client and server, which both run on the same physical machine with one processor. So, he sets out to design a *local remote procedure call (LRPC)* system using the kernel interface from the class notes. (You can find this interface in Table 2-1 in Section 2.D.11, but you should be able to make it through the question without having to consult the notes.).

The kernel interface is implemented by the kernel program, which runs in the kernel address space. The kernel manages the address spaces and threads. The client and the server each run in their own user address space. The client and server each start with one thread. There are no user programs other than the client and server running on the machine.

Communicating messages. To implement LRPC, the client and server share a block of memory through which they communicate requests and responses. This block is at physical address *WELLKNOWNBLOCK* and contains a *request* and a *response*, each of the type *structure message*[1]:

```
structure message {
   char data[1024];  // the data of the marshalled message (at most 1024 bytes)
   int size;         // the size of the data
   bool present;     // is a message present?
};

structure message request;  // request is an instance of structure message
structure message response;  // response is an instance of structure message
```

The client and server both map the block of memory at physical address *WELLKNOWNBLOCK* into their address spaces. The client maps it at the virtual address *CLIENT* and the server maps it at the virtual address *SERVER*.

Server operation. To prepare for receiving an LPRC, the server registers with the kernel a virtual address (*STACK*) to be used as stack pointer and an entry point (the virtual address *receive*, where the procedure RECEIVE is located). The server maps the well-known block in its address space *server_asn* (the identifier for the server address space). Then, it calls the supervisor call YIELD so that YIELD can schedule another thread, which might perform an LRPC to the server.

```
procedure SERVER()
 $register\_gate$(STACK, $receive$);     // register entry point; see below
 $allocate\_block$(WELLKNOWNBLOCK);     // allocate block of physical memory
 $map$($server\_asn$, WELLKNOWNBLOCK, SERVER);     // map it at SERVER
 while ($true$)  do $yield$();
```

[1]Read the code in this section II carefully; it will take time, but you should have enough time.

Name:

When a client thread performs an LRPC, the kernel starts the LRPC at address *receive* in the server's address space, invoking the procedure RECEIVE, and running with the server's stack at virtual address *STACK*:

```
procedure RECEIVE()
  $do\_RPC(SERVER.request.data, SERVER.request.size, SERVER.response)$;
  $SERVER.response.present \leftarrow true$;
```

The notation *SERVER.request.data* means the named data field (*data*) of the named structure (*request*) located on the page with the named virtual address (*SERVER*).

The procedure RECEIVE invokes the procedure call DO_RPC, which takes the request message as an argument, unmarshals the message, processes it, and when it returns it has filled in the response message with the marshalled result. After DO_RPC returns, the procedure RECEIVE sets *SERVER.response.present* to true, indicating to the client that the response is available. The exact way in which RECEIVE returns to the client is not specified in this problem but you may assume it works correctly.

4. [10 points]: In which address space is the code that implements the supervisor call YIELD located (i.e., the code that schedules the processor)?

(Circle best answer)

[✓ **A.** The kernel address space]

B. The client address space

C. The server address space

D. In all three.

Even though the call to YIELD *may start out in user (that is, client or server) address space, by the time it gets to the code that schedules the processor it must be in kernel mode in the kernel address space, because that is the way that the kernel implementation of the notes chooses to enforce modularity. So the best answer is A.*

Client operation. At initialization time, the client maps the block at physical address *WELLKNOWNBLOCK* to virtual address *CLIENT* in its address space *client_asn*:

```
procedure INIT_CLIENT()
  $map(client\_asn, WELLKNOWNBLOCK, CLIENT)$;
```

To perform an LRPC to the server, the client calls LRPC with the server's address space number *server_asn*, a request buffer (containing the marshalled request) and the request size. LRPC returns a response buffer (containing the marshalled result) and response size:

Name:

```
procedure LRPC($int\ asn, char\ request[], int\ request\_size, char\ response[], int\ response\_size$)
  $copy(request, CLIENT.request.data, request\_size)$;
  $CLIENT.request.size \leftarrow request\_size$;
  $CLIENT.request.present \leftarrow true$;
  $CLIENT.response.present \leftarrow false$;
  $transfer\_to\_gate(asn, receive)$;
  while $(CLIENT.response.present \neq true)$ do {
      $yield()$;
   }
  $copy(CLIENT.response.data, response, CLIENT.response.size)$;
  $response\_size \leftarrow CLIENT.response.size$;
```

The client copies the bytes from the request buffer into the part of its address space corresponding to *WELLKNOWNBLOCK*. (The procedure COPY(SRC, DST, LEN) is a procedure in the client's address space that copies len bytes from *src* to *dst*). Then, it calls TRANSFER_TO_GATE to transfer control to the code at the address *receive* in the server's address space. If the server hasn't generated a response, it calls the supervisor call YIELD to yield the processor. Once the response message is available, the client returns the result (*response* and *response_size*) to the caller of LRPC.

5. [10 points]: For Ben's LPRC to work correctly must the virtual addresses *SERVER* and *CLIENT* have the same value?

(Circle best answer)

[✓ **A.** No, as long as the addresses *SERVER* and *CLIENT* in both address spaces translate to the same physical block *WELLKNOWNBLOCK*.]

B. No, the virtual addresses can map to any physical address.

C. Yes, because otherwise *SERVER* and *CLIENT* will translate to different physical addresses.

D. Yes, because otherwise there is no fault isolation.

Correct: A. For Ben's LRPC to work correctly, both the server and the client must be able to share the physical address WELLKNOWNBLOCK. Since the server and the client each have their own independent page maps, the virtual addresses SERVER and CLIENT can be different, yet by proper setting of the two page maps both virtual addresses can map to the same physical address. Answer B has it backwards; it is exactly because SERVER and CLIENT can map to any physical address that both can map to WELLKNOWNBLOCK. Answer C would be true for a direct-mapped cache design, but it is not true for a virtual memory with page maps. In a page map the value of the virtual address is not constrained by the value of the physical address. Answer D is bogus; the fault isolation depends on how all of the other pages are mapped, as well as how defensively the client and server interpret the shared page.

Implementing gates. Since Ben cannot find the code for the supervisor calls REGISTER_GATE and TRANSFER_TO_GATE in the class notes, he implements them as follows. First, the kernel maintains a structure for each address space in the array *entrypoint*:

Name:

```
structure entry {
   int stack;  // value of stack pointer
   int entry_addr; // virtual address of entry point
   int pmar; // page map address register
} entrypoint[10];   // this kernel manages at most 10 address space
```

In this structure the kernel stores for each address space the address of the stack (*stack*) to be used to invoke a procedure at the address *entry_addr*. The structure also contains the *pmar* for each address space. Except for the kernel address space (number 0), all *pmar* entries in *entrypoint* have the user-mode bit switched on and interrupt-enable bit switched on.

The supervisor call REGISTER_GATE registers with the kernel the virtual address of the stack to be used for calls to the procedure stored at virtual address *addr*:

```
procedure REGISTER_GATE($stack, addr$)
 $entrypoint[current\_asn].stack \leftarrow stack$;
 $entrypoint[current\_asn].entry\_addr \leftarrow addr$;
 $entrypoint[current\_asn].pmar \leftarrow current\_pmar$;
```

The variable *current_asn* and *current_pmar* refer to the user address space and its page map address register that made this supervisor call. In the case of the server, *current_asn* is equal to *server_asn* and *current_pmar* is equal to the server's pmar.

The supervisor call TRANSFER_TO_GATE transfers the processor to the virtual address *addr* in address space *asn*. The supervisor call first checks if *addr* is an entry point that is registered with the kernel, and if so, it pushes the address on the stack registered with the kernel:

```
procedure TRANSFER_TO_GATE($asn, addr$)
 if ($entrypoint[asn].entry\_addr = addr$) then {
     $R0 \leftarrow entrypoint[asn].pmar$;
     $R1 \leftarrow user\_to\_kernel(asn, entrypoint[asn].stack)$;    // translate stack into kernel's address space
     $(R1) \leftarrow addr$;    // push addr (4 bytes) on stack in user address space
     $SP \leftarrow entrypoint[asn].stack$;
     $SP \leftarrow SP - 4$;    // lower stack pointer 4 bytes
     $jmp$(LEAVE);
  }
 else $return$(ERROR);
```

You may assume that registers R0 and R1 are available for use by the kernel.

Then, TRANSFER_TO_GATE jumps to the *LEAVE* stub (copied from the class notes):

Name:

```
LEAVE:
  MOVE R0, PMAR   // move the content of R0 into the PMAR register,
                  // which may result in an addres space switch
  RTE             // pop return address and load it into the PC register
```

As described on page 2-47 of the notes, *LEAVE* sets the *pmar* of the processor to the page table of the address space *asn*. As described in the notes, *LEAVE* is mapped in each address space at the same virtual address.

6. [10 points]: During the execution of the procedure RECEIVE how many threads are running or are in a call to YIELD in the server address space?

(Circle best answer)

A. 0

B. 1

[✓ **C.** 2]

D. 2 or more

The correct answer is C: 2 threads. Let's count them: the server's initial thread is blocked in a call to yield in the procedure server (). The client thread is running in the procedure receive () in the server address space (it got there via the TRANSFER_TO_GATE call). Since the system does not create any new threads after the initial two, we've accounted for all of them.

7. [10 points]: How many supervisor calls could the client perform in the procedure LRPC?

(Circle best answer)

[✓ **A.** 1]

B. 2

C. 3

D. 4

E. 2 or more

The answer is A: 1 call. The only superviser call that LRPC makes is a call to TRANSFER_TO_GATE. Don't be fooled by the possibility of the client calling YIELD one or more times. The code for LRPC includes an invocation of YIELD within a while loop but the client thread will never enter the while loop since the server program sets *CLIENT*.response.present to true before returning.

8. [10 points]: Ben's goal is to enforce modularity. Which of the following statements are true statements about Ben's LRPC implementation?

(Circle ALL that apply)

A. The client thread cannot transfer control to any address in the server address space
The use of "any" in the question was confusing; as a result the question was not considered for grading. We intended to ask if the client can transfer control to an arbitrary address in the server address space. The answer to that question is false for two reasons. First, the client can only end up at the address it registered with the kernel. Second, the client doesn't transfer control on its own: only the kernel can transfer the thread to another address space

B. The client thread cannot overwrite any physical memory that is mapped in the server's address space.
The client program is running in the server address space when executing the RPC and can overwrite any memory in that address space.

C. After the client has invoked TRANSFER_TO_GATE in LRPC, the server is guaranteed to set *SERVER.response.present* to true at some point.
Present might not be set because doRPC might never return. For example, it might issue an illegal instruction or go into an infinite loop.

[✓ **D.**] The procedure LPRC ought to be modified to check the response message and process only valid responses.
An advantage of RPC is that it gives processes the chance to enforce modularity by checking responses for validity. If the client doesn't check the validity of responses, a variety of problems can arise including a client crash or even a malicious server taking control of the client program.

9. [10 points]: Assume that REGISTER_GATE and TRANSFER_TO_GATE are also used by other programs. Which of the following statements is true about Ben's implementation of REGISTER_GATE and TRANSFER_TO_GATE?

(Circle ALL that apply)

[✓ **A.**] The kernel might use an invalid address when writing the value *addr* on the stack passed in by a user program.
The kernel does no checking of addr. A buggy user program might provide an invalid address. Alternatively, the user program might have passed in a bogus stack pointer trying to cause the kernel to store to an invalid address.

[✓ **B.**] A user program might use an invalid address when executing the *RTE* instruction in *LEAVE*.
If a bogus address was passed to REGISTER_GATE *(which then pushed the address on the stack), it will be popped off by the RTE in LEAVE and that RTE is executed in user mode in the user address space.*

C. The kernel transfers control to the server address space with the user-mode bit switched off.
The description of the gate system indicates that all PMAR entries have the user-bit switched on.

[✓ **D.**] The kernel enters the server address space only at the registered address *entry_addr*.
This is precisely the goal of gates: to allow limited access to an address space. Ben managed to get this right.

Ben modifies the client to have multiple threads of execution. If one client thread calls the server and DO_RPC calls YIELD, another client thread can run on the processor.

10. [10 points]: Which of the following statements is true about Ben's implementation of LRPC with multiple threads?

(Circle ALL that apply)

[✓ **A.**] On a single-processor machine, there can be race conditions when multiple clients threads call LRPC, even if the kernel schedules the threads nonpreemptively.
Consider the following scenario: client A is running doRPC and sets the shared response.size field to, for example, the size of a buffer it will fill by reading from the network. A now yields the CPU to wait for input to fill the buffer. In the meantime, B issues an LRPC and jumps into the server address space and calls doRPC. B might not yield and would set the shared response.size field to some different value before completing the RPC. When A wakes-up and finishes the RPC it will return with an incorrect response size causing an error in parsing the response (or if the client is naive, a crash).

[✓ **B.**] On a single-processor machine, there can race conditions when multiple clients threads call LRPC and the kernel schedules the threads preemptively.
Both B and C follow from A. The scenerios described in B and C add more concurrency to the system and allow addtional race conditions. For instance two threads on two processors might begin an LRPC at the same time and cause the shared block to become internally inconsistent.

[✓ **C.**] On multiprocessor computer, there can be race conditions when multiple clients threads call LRPC.
See answer to B.

D. It is impossible to have multiple threads if the computer doesn't have multiple physical processors.
Threads are virtual processors. One processor can support many threads.

End of Quiz I

Name: