

ITEE



## What's New:

A few challenging questions have been thrown in. Focus on the ones you can do before trying these. Also think about what you'll need for the assignment.

### 1. Processes

- a. Write pseudocode for the steps given in Figure 4.3 (p 98 of the book) for processing a context switch. Separate out the case of being interrupted and restarting a process (the save and reload boxes, respectively), but do not include the steps for selecting a ready process (the dots between the boxes in the picture). Assume that the **program counter (PC) is at a known location** you can refer to as a variable after an interrupt (it would actually be on a stack in most real machines).

*Notes: the PC can't be retrieved directly because at this point, the CPU has already moved on to a new instruction, so you need to retrieve the PC from the place it's stored (see text highlighted in question), so:*

```
copy registers including stack pointer to PCB
copy PC from saved location to PCB
hand control to scheduler
approximately reverse sequence for "reload"
```

### 2. Threads

- a. An operating system only has one thread in the kernel. When a blocking I/O request occurs, explain why user-level threads in the process containing the I/O call will not be able to continue, even if they do not depend on completion of the I/O.  
*The operating system doesn't know about user-level threads (it is only aware of processes). So when an I/O call comes through, the entire process is moved onto the waiting queue.*

- b. Win2000 has a *one-to-one* mapping between kernel and user-level threads. Discuss advantages and disadvantages of this arrangement.

*Advantages:*

*blocking calls don't stop the entire process  
the scheduler can allocate time more efficiently  
individual threads can be allocated to different processors in multiprocessor systems*

*Disadvantages:*

*the kernel must intervene in thread switches (slower than user-level threads)  
more work for the kernel in maintaining thread information  
doesn't adapt to different situations (e.g. the way Solaris does)*

- c. Explain the advantages of the Solaris approach of a flexible mapping between user-level and kernel threads.

*Advantages:*

*best of both worlds (fast user level threads & the ability to execute blocking calls without stopping the entire process)*

*Disadvantages:*

*there are a finite number of resources (Light-Weight Processes & kernel level threads)  
scheduling is more complex*

- d. Explain how user-level threads can be implemented (including setting a time limit on any thread's execution) without kernel intervention, except the use of system calls.

*Key ideas here: need ability to set OS timer (the only significant system call) and set up an event handler for when a timeout occurs. Also need to assume the OS will save the state of the process when a timeout occurs in a way accessible to a user-level process, so it can save that state and restore it when the scheduler wants to restart the interrupted thread.*

### 3. Interprocess Communication

- a. The bounded buffer solution on p 109 of the book allows at most `buffer_size - 1` entries of the buffer to be used. Develop a solution which allows the entire buffer to be used.

**General comment:** *the obvious solution is to use a count of the filled spaces but the difficulty is that any straightforward solution will result in a synchronization problem, as both processes will need to update the count.*

*Here's an idea which may work, using 2 new counters, filled and emptied, both initially 0 (max is the number of slots in the buffer, or the number of buffers, in the book's terminology):*

*producer*

```
while (filled - emptied == max) ; // spin
produce // will require some of previous version's code
filled++
```

*consumer*

```
while (filled - emptied == 0) ; // spin
consume // will require some of previous version's code
emptied++
```

**Details missing:** *fill in the details of the accesses `buffer` and the updates of the `in` and `out` variables, and check the solution for correctness. The key idea in this algorithm is the fact that there is no race to update the counter, since each process updates a different counter.*

**Another question:** *is it worth going to this trouble? Assume the `buffer` array is an array of pointers, and a pointer is the same size as an integer. Are we ahead in terms of total memory used? What will the values of `filled` and `emptied` be after a large number of uses of the buffer? Does this present a problem, and if so, how could it be solved?*

- b. Differentiate between a mail box owned by the operating system, versus one owned by a specific process. Consider efficiency, convenience and potential correctness issues.

*Mailboxes persist between processes, whereas process owned mailboxes are reclaimed after the process dies.*

*The OS must decide which process receives mail if more than one process wants mail from an OS-owned mailbox, whereas the owner of a process-owned mailbox is always the recipient.*

*If a process owns a mailbox, only that process should be able to receive messages using it, but other processes may be able to send to the mailbox (depending on permissions) – since the goal is to achieve interprocess communication.*

### 3. Programming Issues

Note that the API given here differs in a few details from the Pthreads-based API in lectures.

In each case, given a specific interface to a system call, construct a call, given the following type and function declarations:

```
typedef struct {
    const char * filename;
    const char * filemode;
} FileInfo;

// used to pass lines of text to or from threads

typedef struct {
    char * lines;
    int length;
} CharBuf;

// -----functions-----

void *sortBuffer(void *arg);
void *readFile(void *arg);
void printBuffer (const char message [], CharBuf * buffer);
```

- a. Launch a new thread, passing it the function `sortBuffer` and data to sort in variable `data` of type `CharBuf` using a function with the following interface (note that this is not exactly the same as that in the lectures):

```
int start_thread (void *(*start_routine) (void*), void *arg);
```

Assume the function returns an integer thread ID for later use to identify the thread.

```
CharBuf data;
int newID;
```

```
newID = start_thread (sortBuffer, &data);
```

- b. Wait for the thread to complete, using the following function, which uses the thread ID returned from `start_thread`, and returns a pointer to a result of arbitrary type, and assign the result of the system call to a variable of type `CharBuf`. You will need to do a cast to avoid a compiler warning, since `void *` represents an arbitrary pointer type

```
void * wait (int threadID);
CharBuf * result;
result = ((CharBuf *) wait (newID));
```

4. Do the questions at the end of Chapters 4 and 5 of Silberschatz *et al.*

**General comment:** some of these questions could be useful exercises but don't embark on any which seem to be very difficult, or which look like large projects. I will answer questions on these if they are presented to me. In general, questions at the end of the chapters are "supplementary reading" – I don't guarantee that they are in general good questions, but they could be useful to work through. In future tutorials, I will generally not point explicitly at questions at the end of the chapter.