

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. Since last year's assignment was a bit more complex than this year's some notes on how the 2004 assignment was designed are included.

Learning objectives for this week:

- Understand the major concepts we've covered, including:
  - processes and threads
  - communication
  - user-level vs. kernel-level aspects of processes, threads and communication
- Apply these concepts to reasoning about OS components and implementation, including understanding design choices.
- Apply these concepts to solving programming problems.
- Understand the performance implications, including speed and reliability, of various design options.

As before, you should aim to answer all of these questions by the end of the course. Being able to answer them by the end of week 4 is a useful goal. Be prepared: if you know what you can't do yourself, you will make most efficient use of the tutorial.

## 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).

## 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.
- b. Win2000 has a *one-to-one* mapping between kernel and user-level threads. Discuss advantages and disadvantages of this arrangement.
- c. Explain the advantages of the Solaris approach of a flexible mapping between user-level and kernel threads.
- 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.

## 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. (A bit of a challenge.)
- 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.

## 4. Programming Issues

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.

- 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);
```

## A few C Reminders – For Assignment 1

Headers and Libraries

Unlike Java, C does not have a system of recording types of packages. Instead, *header files* are used in a loose attempt at maintaining consistent types across separately compiler portions of a program.

A header file is imported to a file being compiled using a *preprocessor directive*, which has the effect of including the contents of the header file at the point where the directive appears, as if it had been part of the original file. If you include a system header file (usually stored at `/usr/include` on a UNIX setup, but a C compiler is generally configured to know the default location), its name appears in angle brackets. A header file which you create (searched for at the current directory) has its name in double-quote symbols, e.g.,

```
#include <stdio.h> // printf, etc.
#include "fileTypes.h"
```

It's possible to tell the compiler to look for headers at locations other than the defaults.

Library files contain precompiled code. Again, for simplicity, assume we only need libraries in the default locations. You tell the C compiler (actually the linker) which libraries you need using the `-p` command-line option on a UNIX system. For example, `-pthread` on a Solaris machine means link the library `/usr/lib/libpthread.so` (the detail of the translation from the linker name to an actual file name differs from system to system). Look at the "compile" file supplied with the 2004 assignment to see which libraries have been used. Compare that with the 2005 assignment.

It's possible to compile a C program one file at a time, then link all the compiled files and libraries in a separate step. However, with a small program, it's not worth the effort – in Assignment 1, the given compile command does everything in one line.

### Pointers

Pointers in C are a lower-level equivalent of references in Java. A pointer in C is just a number (usually stored in the same sized unit as an integer, representing a location in memory). It is possible to allocate new memory in C using `malloc`. The result returned by `malloc` (`sizeToAllocate`) is an untyped pointer (for which there is a special type, `void*`) pointing to a newly allocated area of memory of the specified type. Assigning a value of type `void*` (which you read as "pointer to no specific type" in C, writing "\*" after a type name makes it a pointer type) will result in a warning but not an error message from a C compiler. To get rid of the error message, you need to use a *cast* – an instruction to the compiler to override the default type.

For example, if you want to allocate a buffer of characters able to contain up to 64 characters, you would do it like this:

```
char * buffer; // define buffer as a variable of type pointer to char
buffer = (char *) malloc (sizeof (char) * 64);
```

To *dereference* a pointer variable, i.e., access the value at the location the pointer points to, use "\*" before the variable name, e.g., assuming values have been stored in the buffer, you can access the first character by

```
*buffer
```

The variable `buffer` however contains 64 characters. How can we access e.g. the character numbered 42? The first character is at position 0, and C allows us to do pointer arithmetic, so we can access the character at position 42 as

```
*(buffer+42)
```

Equivalently, we can write

```
buffer [42]
```

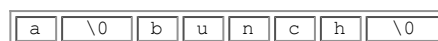
which may be a somewhat familiar notation. In fact, arrays are pretty much the same thing as pointer-based data in C. Bizarrely, since the two notations are equivalent, we can equally well write the following two equivalent expressions:

```
*(42+buffer)
42[buffer]
```

The "\*" operation has the effect of converting a pointer value to the actual value the pointer is pointing to. Sometimes, a pointer is needed when the value you have is not a variable created through a pointer. In that case, you can make a pointer to an existing value (generally, a variable, but it can also be an element of an array or other data structure) using the "&" operation. For example, let's say we would like to pass a pointer to item 42 in our by-now familiar buffer to a function. We could create the necessary pointer like this:

```
&buffer[42]
```

Why would we want to do this? The buffer may represent several arrays concatenated together to save memory. A *string* in C is represented as an array of characters, of which any up to but not including a special character '`\0`' form part of the string. More than one string can be packed into an array, since anything from the '`\0`' onwards is ignored, e.g. the following array, `buffer`, with 8 elements:



represents 2 strings, one starting at position 0, the other at position 2. We can print the two strings as follows

```
printf ("string 1 = %s\nstring 2 = %s\n", buffer, &buffer[2]);
```

Check through the source code supplied with the assignment, and see how much of this you can identify. Also look at the solution supplied for the assignment of 2004 for more examples.