**Amdahl's Law**
If 40% of the code can be parallelized (i.e., run in two threads) how much faster will the code be?

\[
\text{speedup} = \frac{\text{old}}{\text{new}} = \frac{1}{(1 - F) + \frac{F}{S}}
\]

where \( F \) = fraction of code that is enhanced, \( S \) = speedup of enhanced code

\[
1 / ((1 – 0.4) + (0.4/2)) = 1 / (0.6 + 0.2) = 1 / (0.8) = 1.25
\]
i.e., it's 25% faster

what if we used four threads?
\[
1 / ((1 – 0.4) + (0.4/4)) = 1 / (0.6 + 0.1) = 1 / (0.7) = 1.43
\]

theoretical maximum speedup
\[
1 / ((1 – 0.4) + (0.4/\text{INF})) = 1 / (0.6 + 0) = 1 / (0.6) = 1.67
\]
so you can only make it a maximum 67% faster

**Threads**
- independent lines of execution in same program
- have their own PC and stack; share global & heap

Threads are not actually part of the C programming language, but the POSIX Threads (“pthreads”) library is widely supported and widely used.

```c
#include <pthread.h>

// this is the function that will be called when the thread starts
// it MUST match this signature!
void *fun(void *);

main() {
    int fun_arg = 10; // argument to pass to function
    pthread_t thread_id; // id of thread that will be created
    // create/start the thread
    pthread_create(&thread_id, NULL, &fun, &fun_arg);
    ...
    // wait for the thread to finish
    pthread_join(thread_id, NULL);
}

// function that is called when thread starts
void *fun(void *p) {
    // remember to cast the void pointer before dereferencing it!
    int y = *(int *)p;
    printf("The value is \%d\n", y);
}
```
Things to look out for:
- you're passing a pointer to pthread_create, so be careful that some other thread doesn't change it
- if “main” finishes before the thread does, the program will end, so be sure to use pthread_join to wait for it

A pretty decent tutorial: [http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html](http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html)

What can go wrong when using threads?
Race conditions: one thread modifies a variable being used by another (in a bad way)

The famous bank account example!

```c
double balance;
double deposit(double amount) {
    balance += amount;
    return balance;
}
```

If two threads attempt to simultaneously execute this code on the same processor, we could have a race condition if one thread affects the other. It seems like it wouldn't happen here (since once the balance is updated, it's okay if there's a context switch there), but think about what this looks like in assembly/machine language:

```
; this is assembly for balance += amount on an x86-like CPU
load r0, amount
load r1, balance
add r0, r1, r0
store r0, balance
```

The processor has no idea that those four instructions “go together”, so it may happen that we switch from one thread to another during their execution.

Let's say the balance is set to 20, and that Thread-1 calls deposit(100) and Thread-2 calls deposit(40):

<table>
<thead>
<tr>
<th>Thread-1</th>
<th>Thread-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>load r0, amount ; r0 &lt;-- 100</td>
<td>load r0, amount ; r0 &lt;-- 40</td>
</tr>
<tr>
<td>load r1, balance ; r1 &lt;-- 20</td>
<td>load r1, balance ; r1 &lt;-- 20</td>
</tr>
<tr>
<td>add r0, r1, r0 ; r0 &lt;-- 120</td>
<td>add r0, r1, r0 ; r0 &lt;-- 60</td>
</tr>
<tr>
<td>switch!!!------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>store r0, balance ; 120--&gt;balance</td>
<td>store r0, balance ; 60--&gt;balance</td>
</tr>
</tbody>
</table>

What happened?? Balance is set to 120, and we lost our deposit of 40.

To avoid this, we can try to make that instruction “atomic” (Java has support for this, C does not) or we
can use “locks”.

That is, we attempt to lock the **critical section** so that only one thread can access it at a time.

In POSIX threads, we use a “mutex”, which has functions for locking (preventing other threads from entering) and unlocking (allowing any waiting threads to proceed).

```c
double balance;

pthread_mutex_t deposit_lock;

double deposit(double amount) {
    pthread_mutex_lock(&deposit_lock);
    balance += amount;
    pthread_mutex_unlock(&deposit_lock);
    return balance;
}

main() {
    ...
    // somewhere before calling deposit the first time
    pthread_mutex_init(&deposit_lock, NULL);
    ...
```

When using locks, avoid common mistakes like:
- Deadlock: one thread has a lock that the other wants, and vice-versa
- Forgetting to unlock
- Unlocking when you don't have the lock

To avoid these problems, you can use a tool called Helgrind. It is part of the Valgrind suite of analysis tools. If your program is called program.c, then just do this:

```
cc -g -o program program.c
valgrind --tool=helgrind ./program
```

This will analyze your program and will print out warnings about possible thread-related problems.