THREADS

Ruben Acuña

Fall 2017

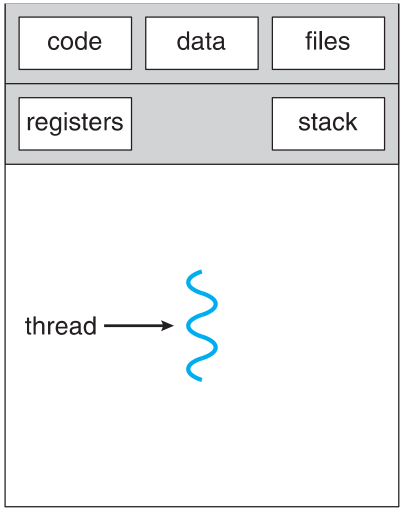
2

OVERVIEW (4.1)

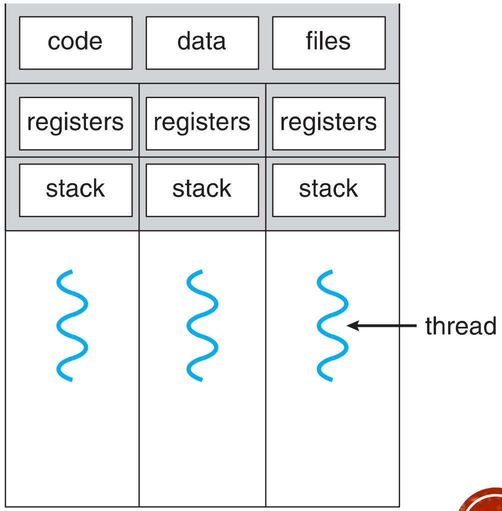
THREADS

- Until now, we've been talk about processes which are effectively independently programs executing concurrently. Each process has one "thread of execution".
- This is a little expensive (why?) if we just need to farm out simple computation.
- So, what if we allowed processes to contain multiple threads of execution over the same data? That is, we had a *multithreaded process*.

- Benefits:
  - Responsiveness (incl. Recoverability)
  - Resource Sharing
  - Economy
  - Scalability



single-threaded process



multithreaded process

Image from Operating System Concepts.

MULTICORE PROGRAMMING (4.2)

MULTI-THREADING

- Parallelism vs Concurrency: parallelism means for two independent threads of execution to continue simultaneously, while concurrency means for them to both be 'alive' at the same time.
  - Really, this doesn't matter to us. Design for concurrency and let the physical hardware do whatever it can.
- Of course, adding extra cores isn't a magic bullet for reducing execution time. Programs typically have some portion that MUST be executed serially (why?), this gives a permanent bottleneck (see Amdahl's Law).
  - Plus: the overhead of creating threads, which means we shouldn't create them unless there is significant work for it.

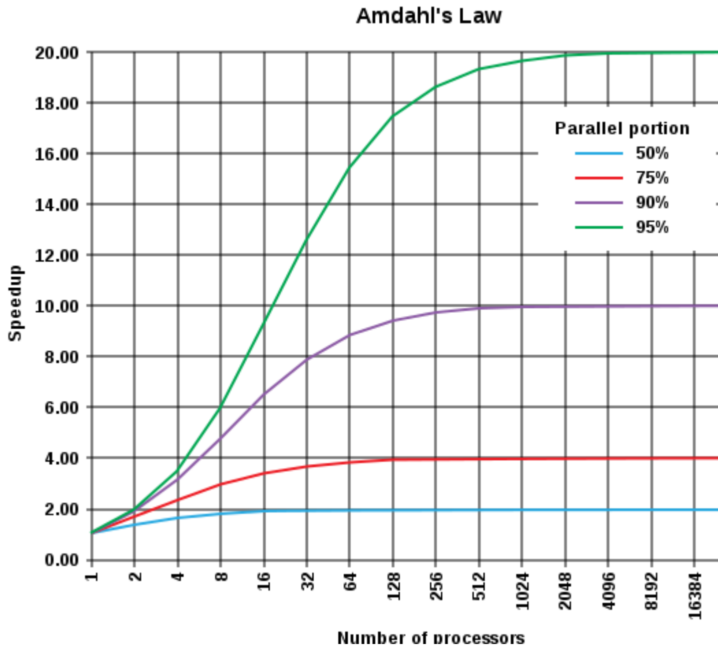


Image from Operating System Concepts.

ISSUES

- Unfortunately, creating a multi-threaded program is hard…
  - Identifying Tasks: Need to identify independent functionality in a program.
  - Balance: If we choose to multi-thread, then each thread should do a comparable amount of work.
  - Data Splitting: Memory is finite and not free to copy! Must identify what data each thread needs and provide only it.
  - Data Dependency: Some tasks cannot be parallelized because there is no redundancy or independence among them – i.e., they enforce serial behavior.
  - Test and Debugging: Normally it is enough to debug a program "at a single line", but here, we must worry about every combination of instruction for each active thread.

PARALLELISM TYPES

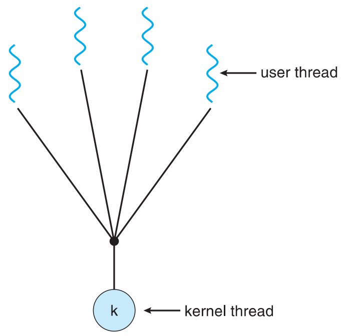
- Data Parallelism: the computation is expressed over a pool of data in such that computation may take place simultaneously on some subset of the data.
  - Recall the divide and conquer algorithm design technique?

- Task Parallelism: the computation is expressed over a pool of data such that different computations may simultaneously take place on the entire data set.

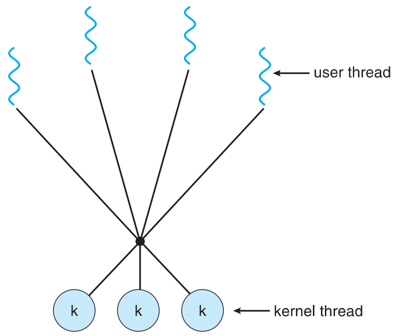
MULTITHREADING MODELS (4.3)

MAPPING KERNEL PRIVILEGES

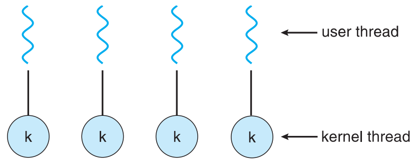
- A thread may operate in either user space or kernel space. Consider: if a thread in user space executes a privileged system call, who executes it?
  - There must exist some mapping between user threads and kernel threads.



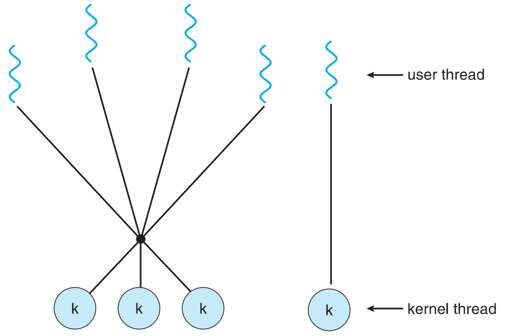
N to 1



N to M



N to N



N to M; 1 to 1

Images from Operating System Concepts.

THREAD LIBRARIES (4.4)

Skipping 4.4.2 (Windows Threads).

OVERVIEW

- Different threading libraries are available on same perform, each may run in user space or kernel space. (If we pick an API that uses kernel space, our previous discussion is moot.)
- Note that since we are using threads, not processes, we don't have the issue of picking a IPC model. Instead, shared memory must be used because the threads do not have individual memory allocations.
- The following examples do summation of 0 to N in a child thread, make the parent process wait synchronously for it, and then display it.

POSIX (PTHREADS)

- Child threads under pthreads have read/write access to global variables (e.g., sum).

```c
#include <pthread.h>
#include <stdio.h>

int sum;
void* runner(void *param);

int main(int argc, char *argv[]) {
    pthread_t tid;
    pthread_attr_t attr;

    //assume argv[1] stores N


    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, argv[1]);

    pthread_join(tid, NULL);
    printf("sum = %d\n", sum);
}
```

```c
void* runner(void *param) {
    int i, upper = atoi(param);
    sum = 0;
    for (i = 1; i <= upper; i++)
        sum += i;
    pthread_exit(0);
}
```

For C++, see Boost::Threads.

JAVA

- Super easy: make a class implement Runnable, then start it with a .start().

```
class Sum {
    public int sum;
}

class Summation implements Runnable {
    private int upper;
    private Sum su;
    public Summation(int upper, Sum su) {
        this.upper = upper;
        this.su = su;
    }
    public void run() {  //from Runnable
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        su.sum = sum;
    }
}
```

- Child threads under Java have read/write access to anything that they can reference. Solution here is a wrapper class.

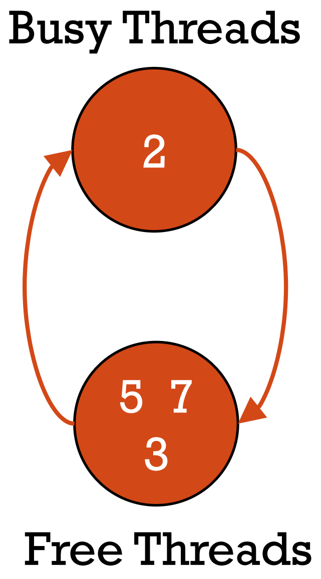
```
public static void main(String[] args) {
    //Assume args[0] stores N.
    Sum su = new Sum();
    int upper = Integer.parseInt(args[0]);
    Thread thrd =
        new Thread(new Summation(upper, su));
    thrd.start();
    try {
        thrd.join();
        System.out.println("sum=" + su.sum);
    } catch (InterruptedException ie) { }
```

Code from Operating System Concepts.

IMPLICIT THREADING (4.5)

THREAD MANAGEMENT

- Threading an application is a hassle!
- Ideally, we'd like for the compiler to do some sort of data flow analysis and identify independent threads of execution, or, induce independent threads of execution by code rewriting. This is an extremely hard problem!
  - A cheaper alternative is to provide a sort of markup language that lets us specify (or hint) at parallelism.
- Without going too far, one simple abstraction we can make is encapsulating the creation and destruction of threads.
  - Remember the factory design pattern?
- We'll call it *thread pool*ing. The idea is to maintain a set of threads in a program that exist but are not executing. The goal is to simplify running a separate task to look something like an RPC.



OPENMP

- Consider the following two code fragments, which sum an array or add vectors respectively:

```
for (int i = 0; i < n; i++)
        sum += d[i];

for (int i = 0; i < n; i++)
        c[i] = a[i] + b[i];
```

- Is any sort of parallelism possible with these examples? How hard would it be?
- OpenMP is a library that supports threading an application by examining “hints” made the programmer as to which structures may be parallelizable. In terms of C, that means adding directives.
  - Conceptually, one may think of OpenMP as providing resource management while the program provides vectorization.
- Note that compilers must include support for OpenMP, or we get nothing!

OpenMP Examples

- Sample Directives:
  - *parallel* – spawns a set of threads based on available processors, and executes region on each of them.
  - *for* – equally distributes for-loop iterations over available threads.
  - *sections* – indicates some set of sections that are independent and may be run in parallel.
  - *section* – marks a region of code.

```
#pragma omp parallel
{
        printf("Test");
}
```

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < 10; i++)
        printf("%d", n);
}
```

```
#pragma omp parallel for
for (int i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

```
#pragma omp parallel sections
{
    #pragma omp section
    { func1();
      func2();
    }
    #pragma omp section
    { func3(); }
    #pragma omp section
    { func4();
      func5();
    }
}
```

GRAND CENTRAL DISPATCH

- OS X also provides a feature similar to OpenMP sections, but integrated as a library instead of directives.
- Different regions of code are specified as *blocks* with the notation ^{code}.
  - Note that this notation works a little like a first-class function.
- Similar to our discussion of processes: blocks will be added to a *dispatch queue* and then are moved to *main* (running) *queue* for execution.
- The abstraction is different than OpenMP, since the number of blocks is invariant with the number of processors.

```
dispatch_queue_t queue = dispatch_get_global_queue(PRIORITY_DEFAULT, 0);
dispatch_async(queue, ˆ{
    printf("I am a block.");
});

dispatch_apply(n, queue, ^(size_t i){
     c[i] = a[i] + b[i];
    });
```

First code snippet from Operating System Concepts.

THREADING ISSUES (4.6)

SIGNALS

- Suppose that an event happens that needs to interrupt of the execution of a thread/process and make the program do something.
- In Linux, we have support for *signals*, which are essentially little flags (e.g., SIGABRT, SIGFPE) that cause a recipient to run a specific function they have assigned for it.
  - Can think of as an enumeration based exception mechanism.

```
//catch signal

void sigint_target(int signum) {
    printf(“Encountered SIGINT.");
}

//bind sigint_target as callback for SIGINT
signal(SIGINT, sigint_target);
```

- Functions:
  - `sighandler signal(int signum, sighandler handler) //also see sigaction`
  - `int raise(int sig)`
  - `int kill(pid pid, int sig)`
  - `int pthread_kill(pthread thread, int sig)`

```
//send signal to current process
raise(SIGINT);
```

Why are signals discussed as thread issues?

SMARTER THREADS

- Thread Cancellation: in addition to manually forcing a thread to terminate by sending a signal, we can also use pthreads to trigger a cancellation on it.

- Thread-Local Storage: sometimes we need the equivalent of a global variable that is unique to each thread.

- Scheduler Activations: Recall user to kernel thread mapping. Internally, the kernel sees threads (and schedules) in user space as being Light Weight Processes (LWP).

OPERATING-SYSTEM EXAMPLES (4.7)

Skipping 4.7.1 (Windows Threads).

THREADS UNDER LINUX

- In addition to pthreads, threads can be created with *clone* (analogous to *fork*).
- Like processes, threads are treated as "tasks", with the main difference being resource sharing. (If nothing is shared clone becomes fork.)
    - Both are represented with task_struct.