

# Operating Systems:

## Threads

### Chapter 2

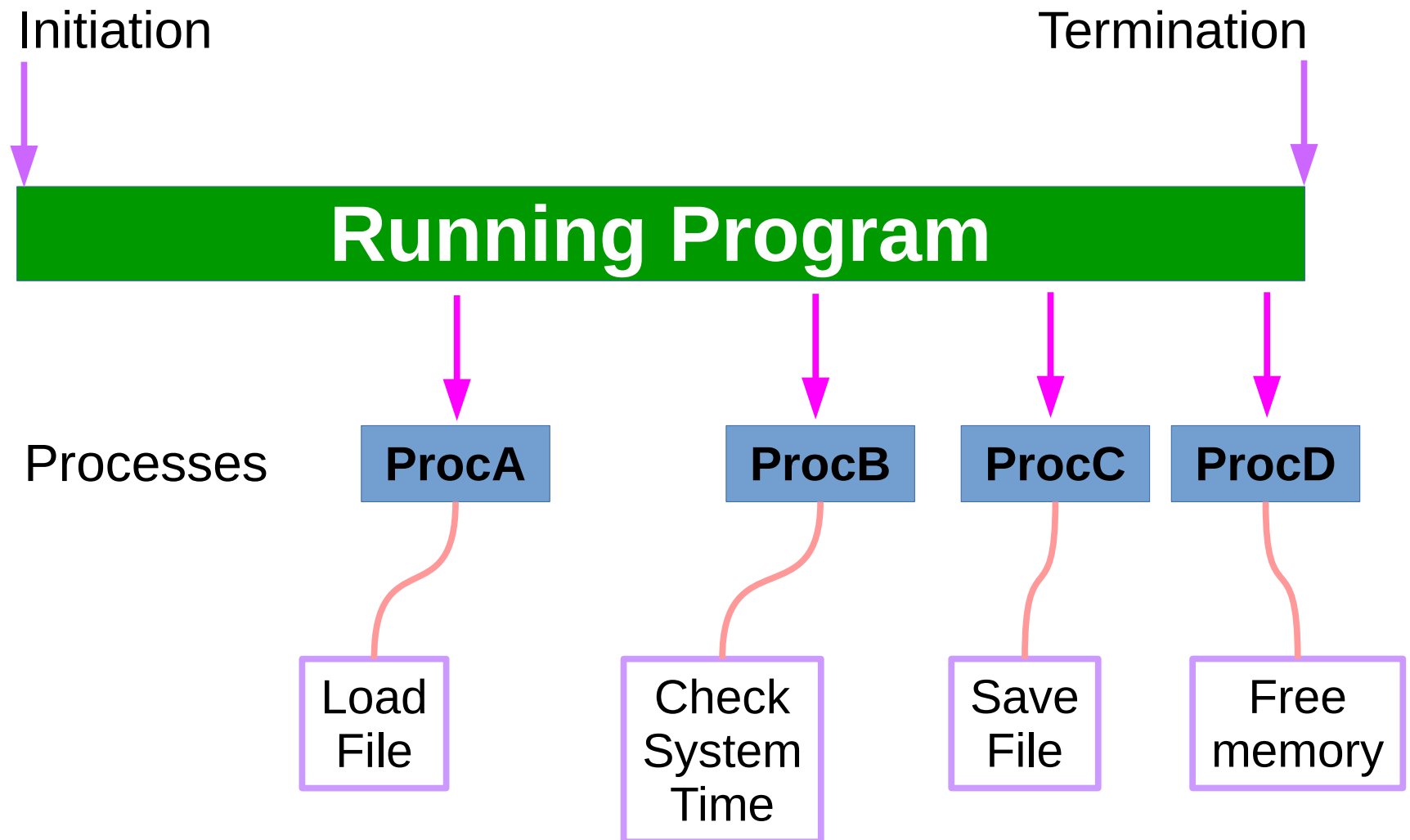
### CS400

Week 3: 28<sup>rd</sup> Jan

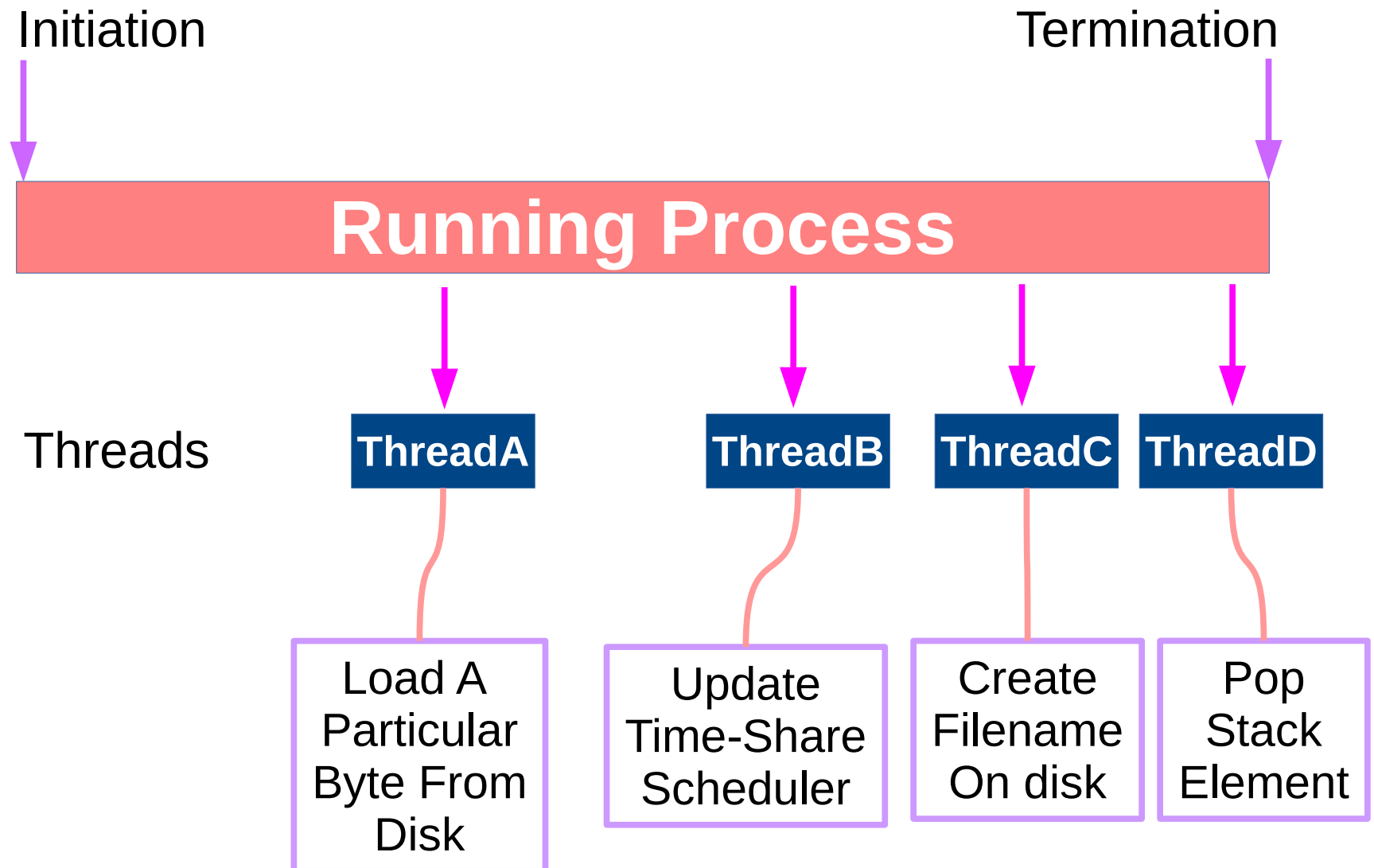
Spring 2020

Oliver BONHAM-CARTER

# Programs versus Processes



# Processes versus Threads



# Arguments for Threads

- 1<sup>st</sup>: Applications have lots of activities happening at same time.
  - Some activities may block and wait for others to complete due to timers, interrupts and etc.
  - Having an application run multiple sequential threads running in quasi-parallel is easier to program, organize in code.
  - Parallel processing from threads: entities can share same memory space and data among themselves.

# Arguments for Threads

- Using threads to handle the application's tasks makes lighter burdens on the system because of ability to share resources
  - Multiple processes (i.e., “*objects*” from *programming*) running do not share memory resources, have separate memory spaces, and may have own resource requirements.
  - Heavy burden on system to manage all these objects...

# Arguments for Threads

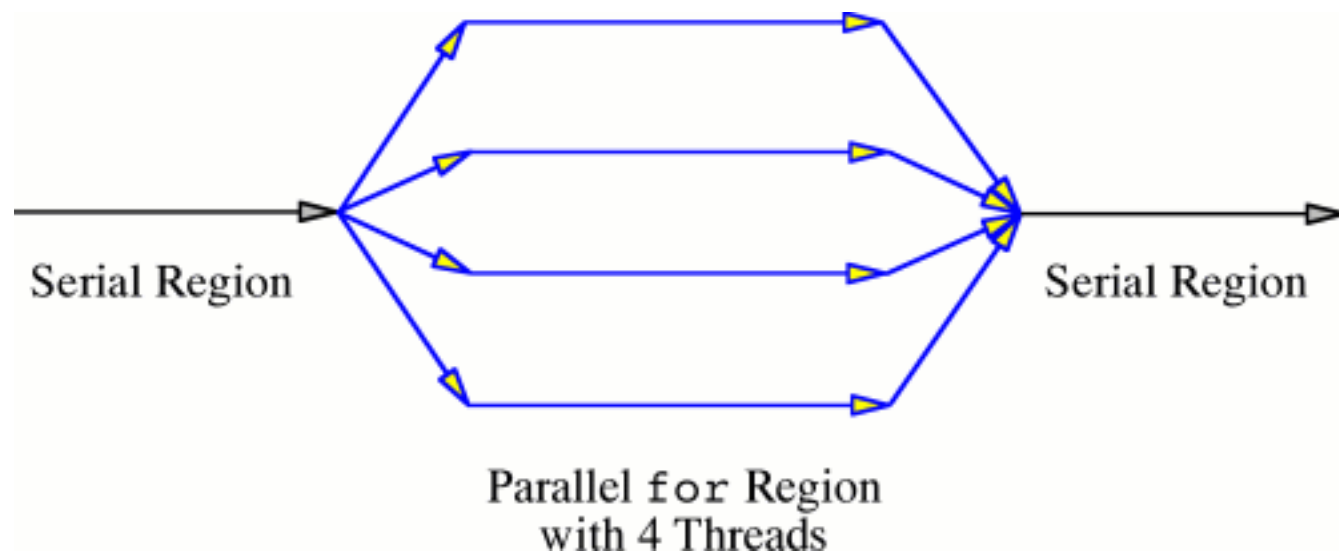
- 2<sup>nd</sup> :Threads are light-weight
- Can be easily (i.e., quickly) created than processes for tasks and are easily destroyed after tasks are completed.
- In many systems, creating a thread goes 10–100 times faster than creating a process.
- When the number of threads needed changes dynamically and rapidly, this property is useful to have.

# Arguments for Threads

- 3<sup>rd</sup> : A performance argument.
- Threads yield no performance gain when all of them are CPU bound, but when there is substantial computing and also substantial I/O, having threads allows these activities to overlap to speed-up the application
- Convenient for machines of multiple CPUs which are able to take advantage of running these threads simultaneously to complete tasks in *real parallelism*.

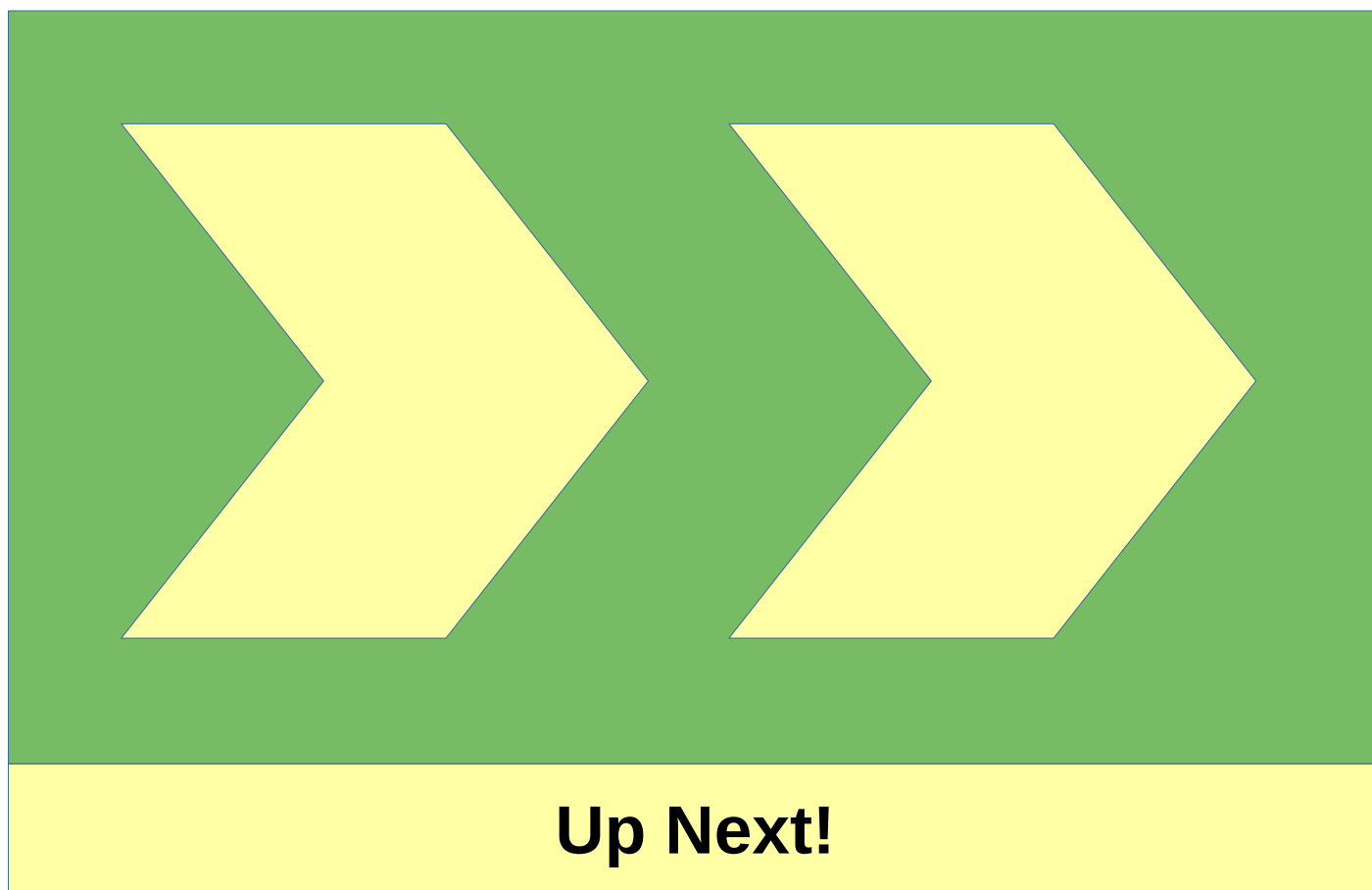
# So, What are Threads?

- Processes that complete “heavy-weight” tasks may need assistance
  - Execution of applications
- Threads offer a help to complete tasks simultaneously
  - Completion of small tasks for to complete the process.



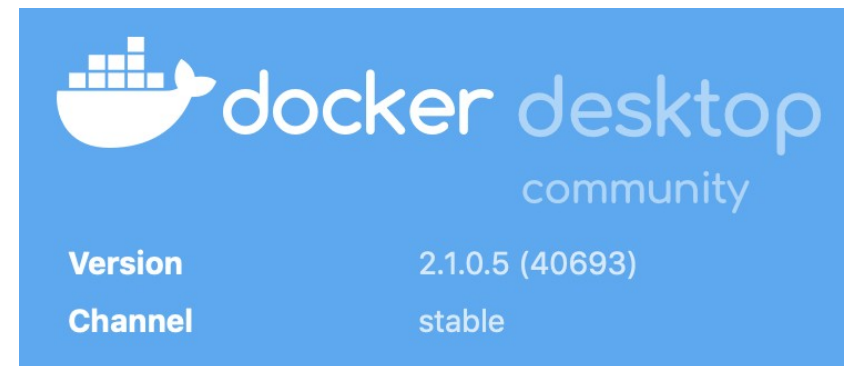


# Let's Code!



# Commands to Run From (Linux) Bash

- Build the container :
  - `docker build -t gccdev .`
- Run the container :
  - `docker run -it gccdev`
- Mount local drive and run container :
  - `docker run -it --mount type=bind,source=$PWD,target=/home/gccdev gccdev`



Note: the directory where you run this becomes your local directory in the container.

# Programming with Threads

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
//Compile try: gcc helloWorldthread.c -pthread

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("\t\t * Cloned thread from number %ld!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("\t In main: creating thread number %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return
                code from pthread_create() is %d\n", rc);
            return 0;
        }
    }
    /* Last thing that main() should do */
    pthread_exit(NULL);
}
```

See file: sandbox/helloThreads.c

**Consider the following**

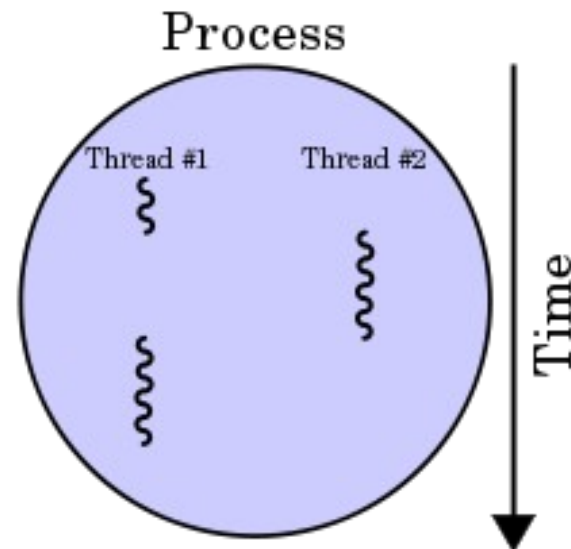
1. What is your output?

2. Can you explain how this output happened?

**THINK**

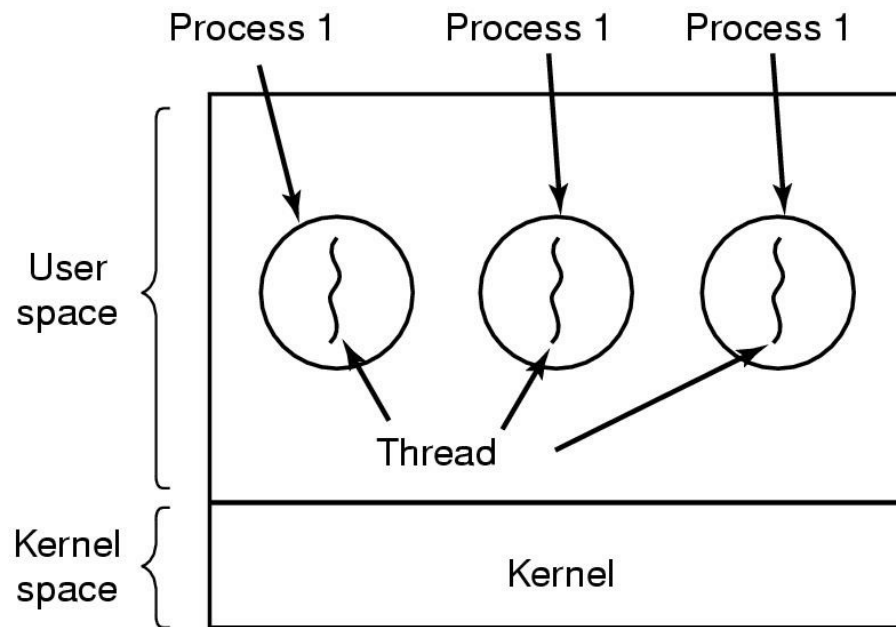
# Characteristics of Threads

- Multiple threads may serve same process
  - Threads share the same **address space** and so are able to communicate between each other (different processes are not allowed to communicate between each other this way.)
  - Threads may read / write to the same data structures and variables which are associated with process.

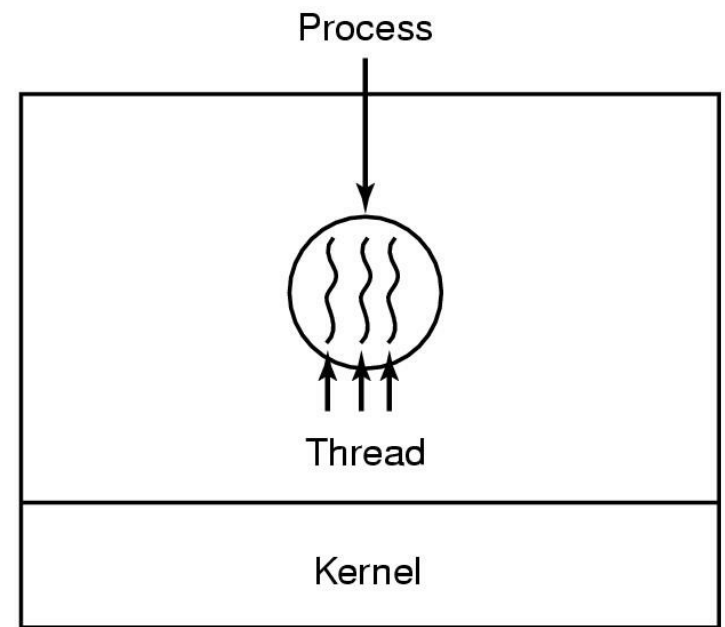


# Models and Usage

- Three processes each with one thread (a)
- One process with three threads (b)
- In (b), we see that while one thread is busy running a process, other threads could be performing other related tasks.

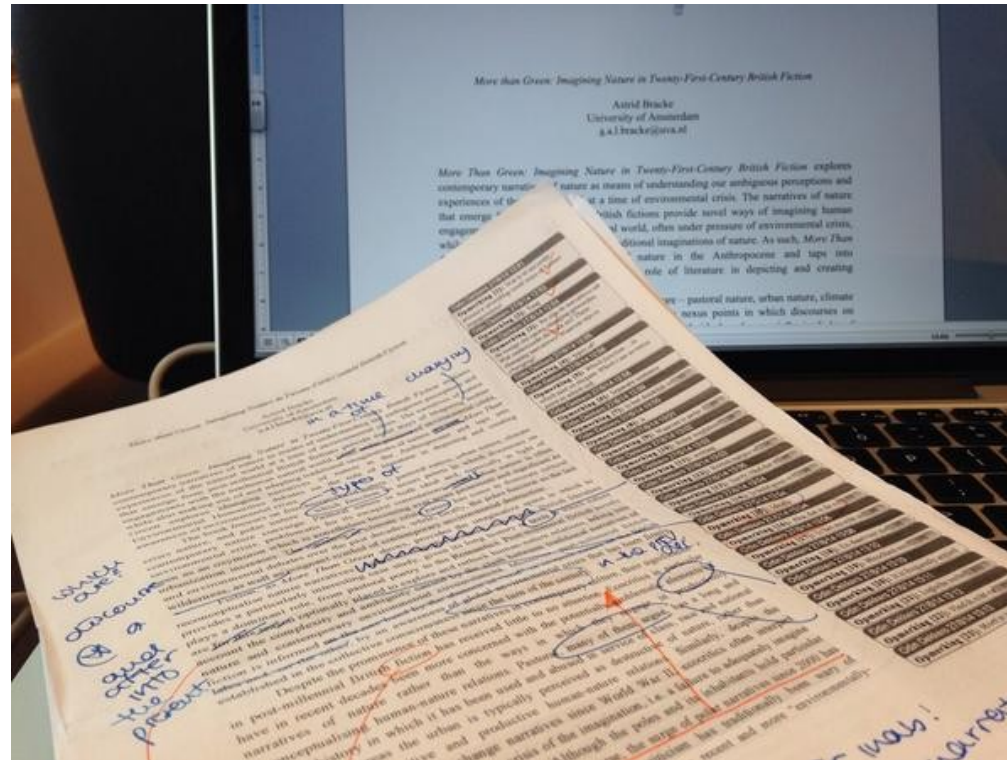


(a)



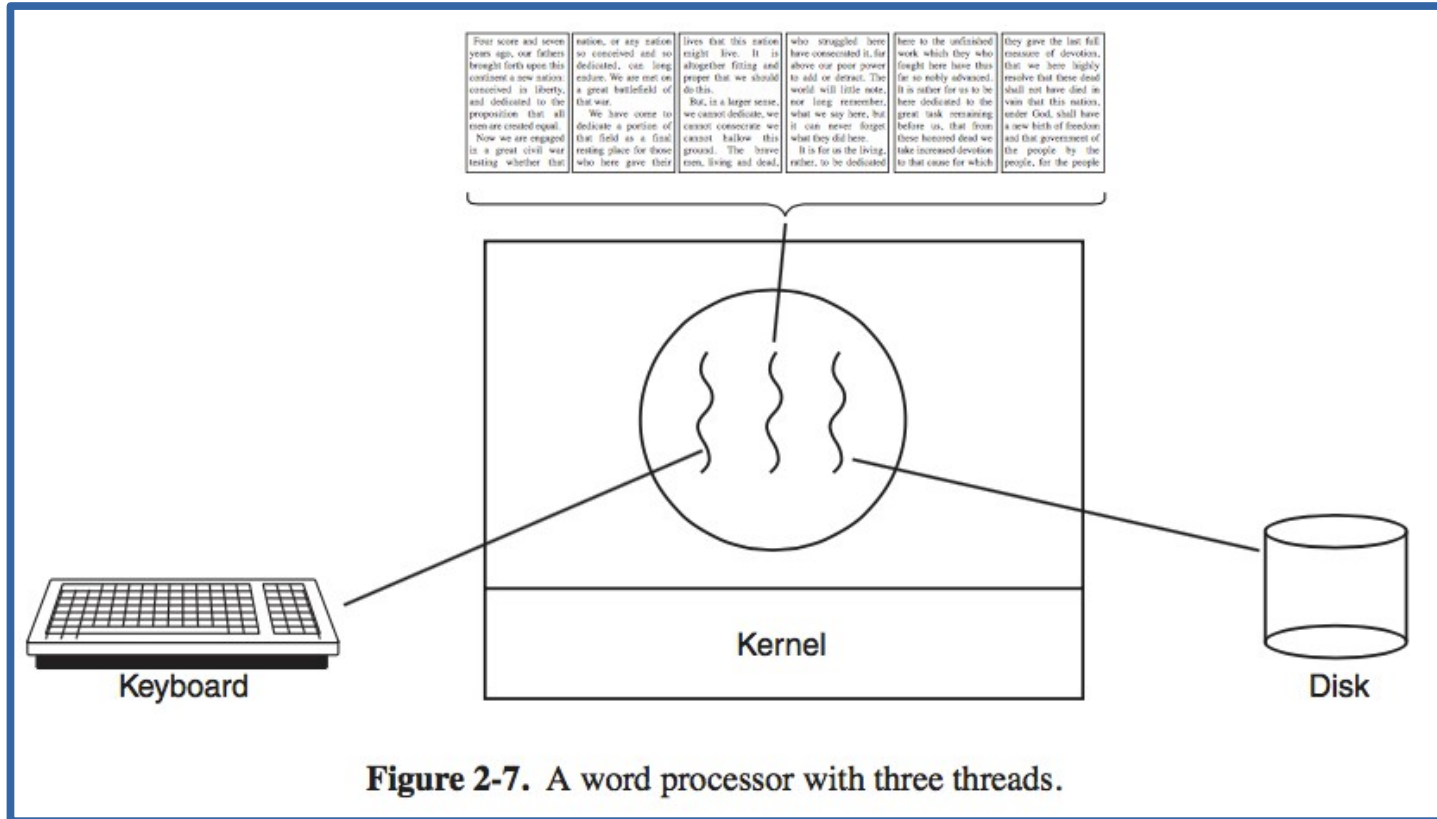
(b)

# Thread Usage Example



- Word processing: Imagine writing a long (600 page) document where first page is removed.
- Removing other pages in the document requires reformatting
- How does the code reformat the doc without making us wait?

# Thread Usage Example

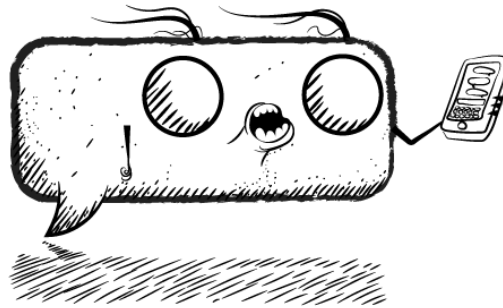


Threads Share common memory and can work together.

- Solution: Have one thread working with user's edits and another (in background) working to reformat the book so that user does not have to wait.
- The third thread handles the editor's functions and recording the inputted text.

# Helpful Threads?

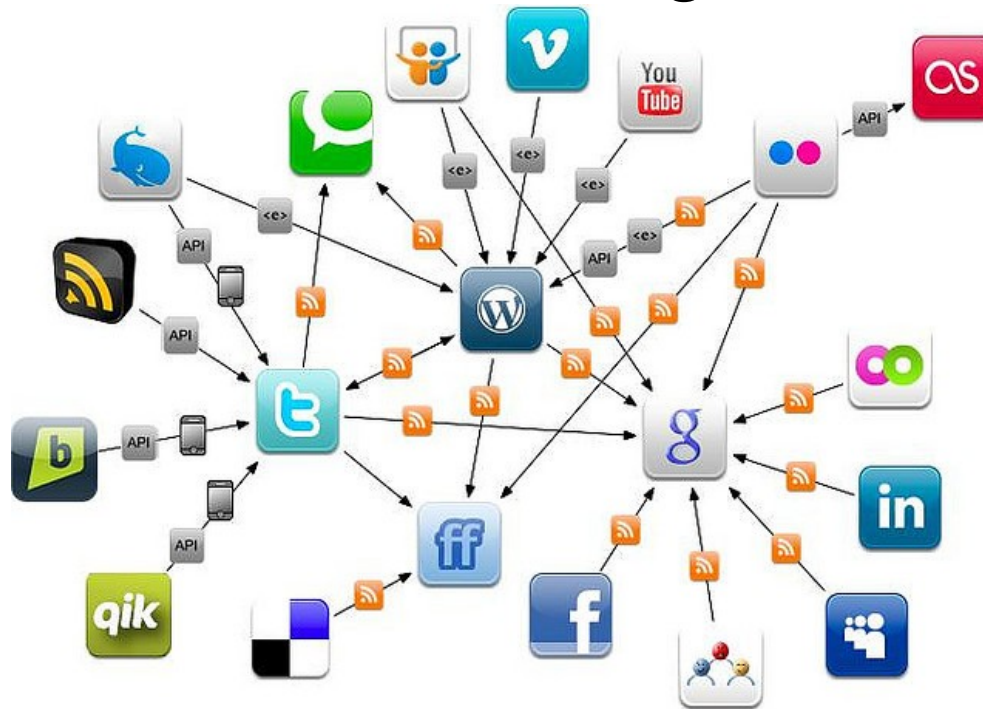
Autocorrect  
hates you



- Threads help with necessary tasks that can happen in the background of computing.
  - Autosaving documents
  - Resizing a browser window
  - Loading and unloading data to a swap file
  - Autocorrect



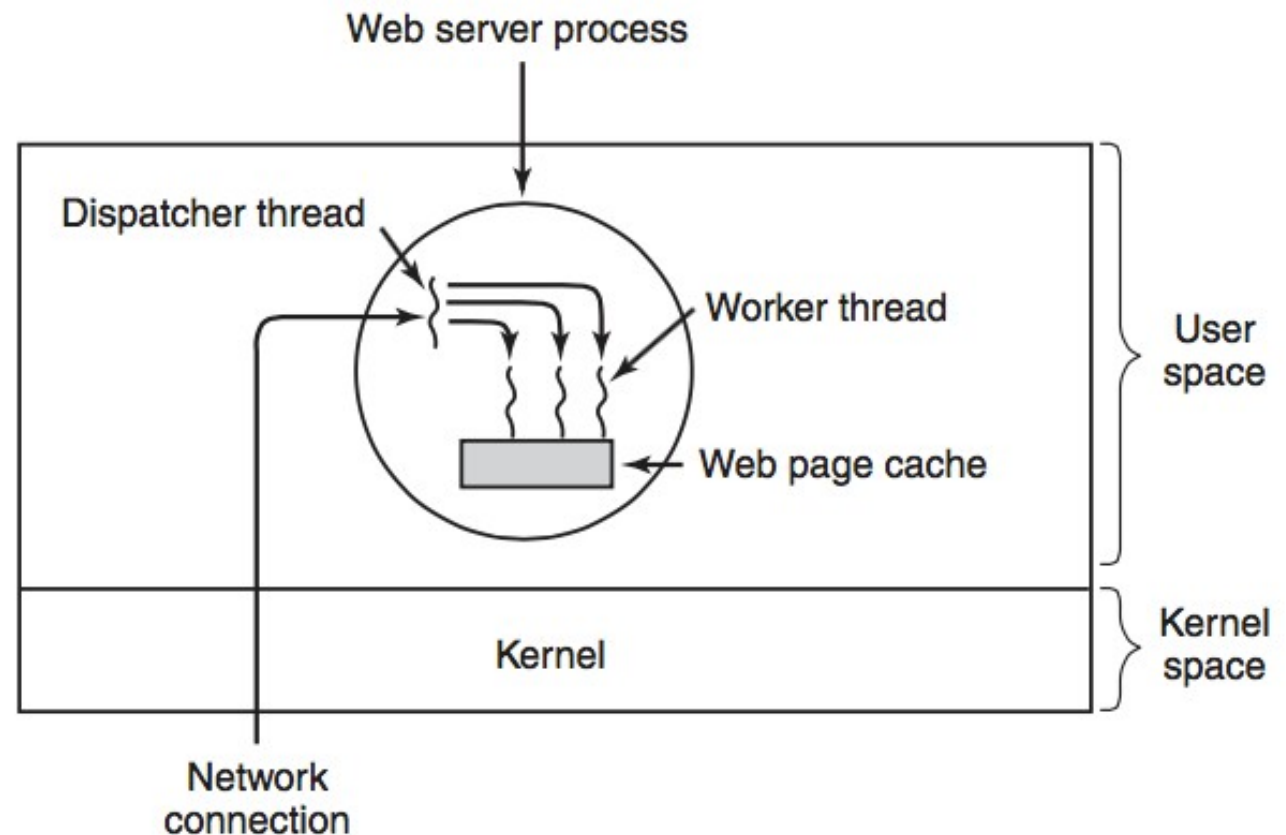
# Threads for Website Management



- On a huge corporate webpage, the main page is more viewed than its pages at the end of a series of clicks.
- Website management software builds threads into code to keep the frequently viewed pages in the computer memory cache (for fast loading).

# Threads to Handle Incoming Requests

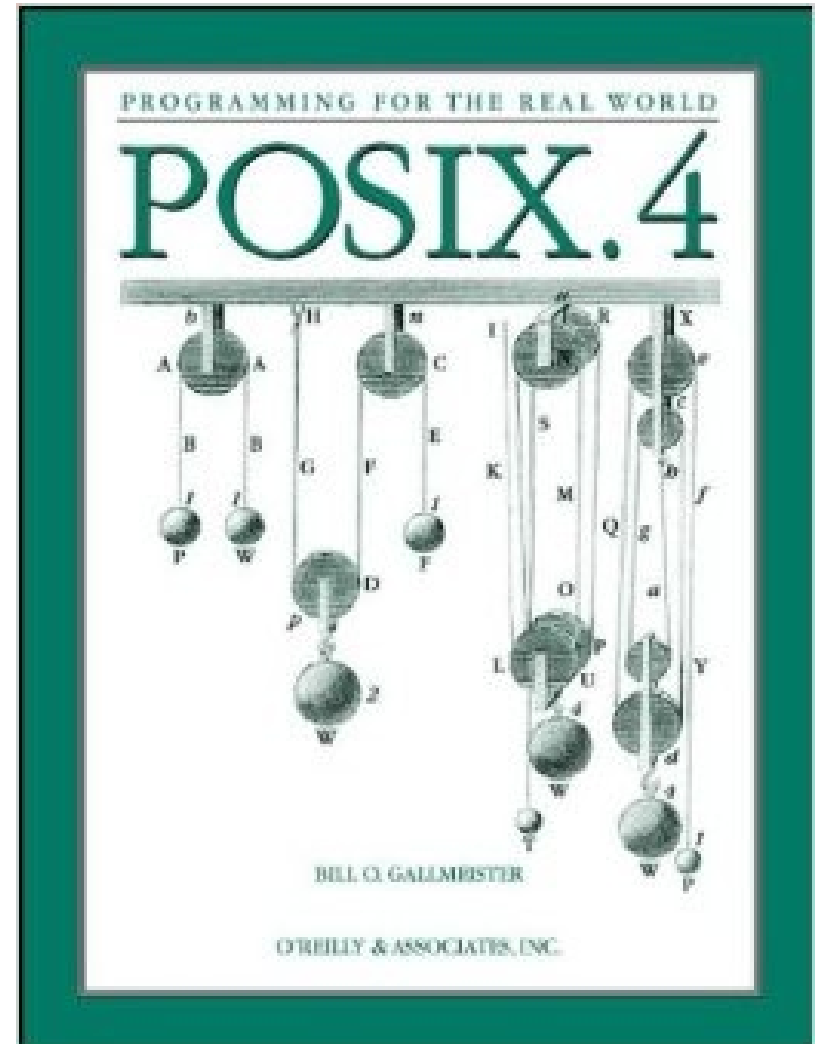
An incoming request is taken by a thread that assigns it to another blocked (i.e., sleeping) thread to be handled.



**Figure 2-8.** A multithreaded Web server.

# POSIX: Standardizing Unix

- OS's use specific-type libraries
  - **Would a Windows driver work on Linux?**
- The Portable Operating System Interface (POSIX): a family of standards specified by the IEEE Computer Society.
- For maintaining compatibility between Unix-type operating systems.
- POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.



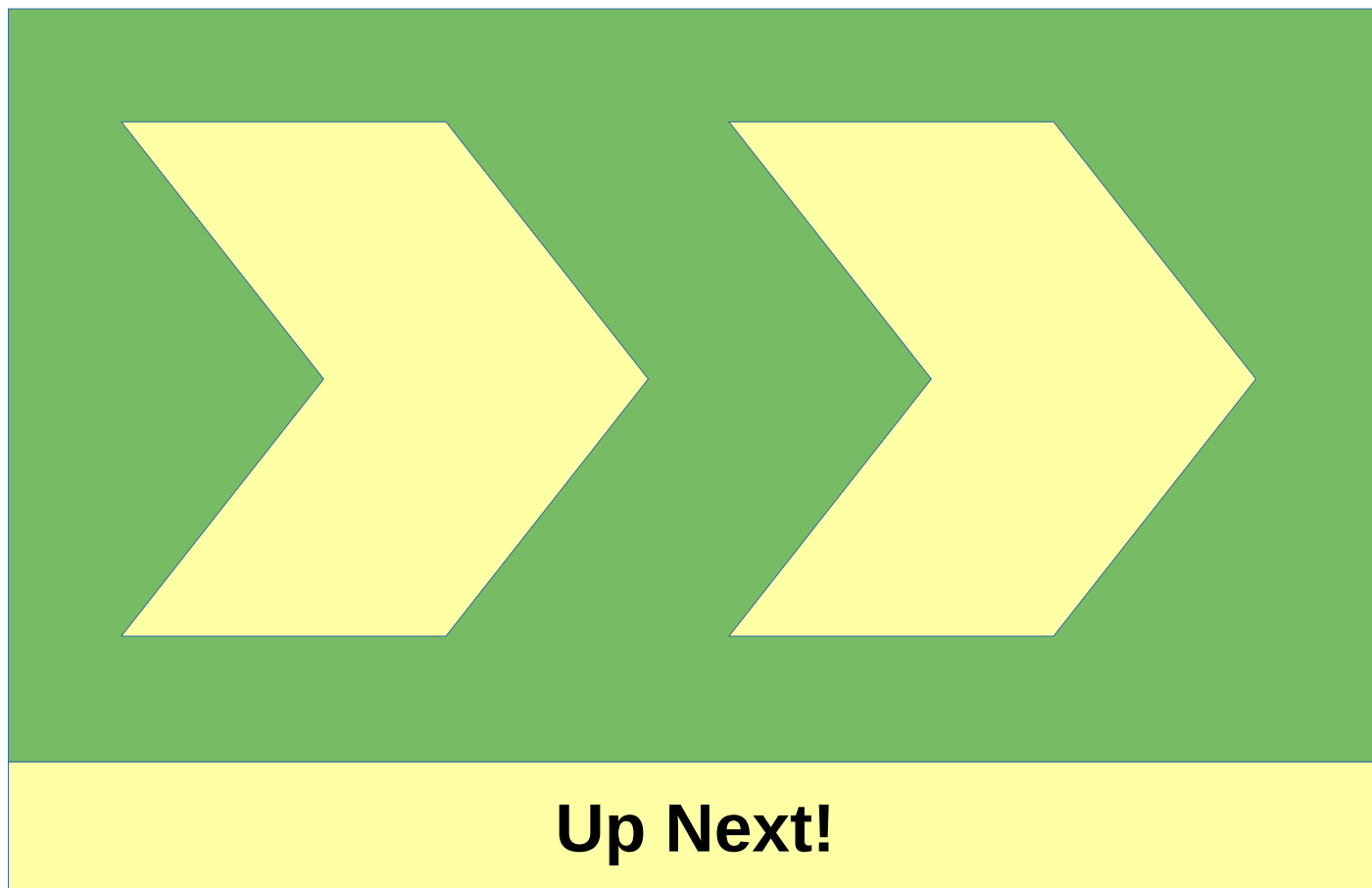
# *Pthreads* – Standardized

- Historically, hardware vendors have implemented their own proprietary versions of threads.
- These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- In order to take full advantage of the capabilities provided by *threads*, a standardized programming interface was required.
- The creation of the ***pthread*** (Posix-thread)

# IEEE and Pthread

- The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification.
- Some useful links:
  - <http://standards.ieee.org/findstds/standard/1003.1-2008.html>
  - [www.opengroup.org/austin/papers/posix\\_faq.html](http://www.opengroup.org/austin/papers/posix_faq.html)
  - [www.unix.org/version3/ieee\\_std.html](http://www.unix.org/version3/ieee_std.html)

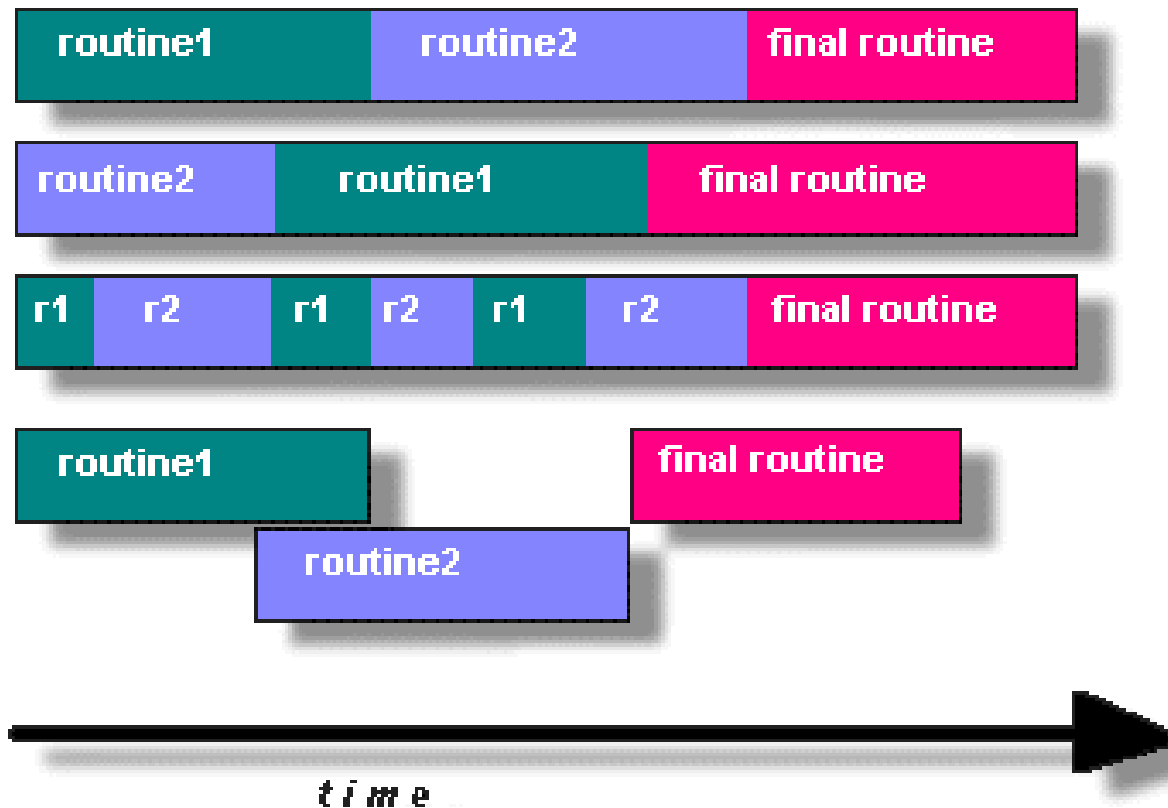
# Let's Code!



See some *pthread* source files in sandbox/

# Using Pthreads

- Pthreads program: made up of discrete, independent tasks which can execute concurrently.
  - For example, if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading.

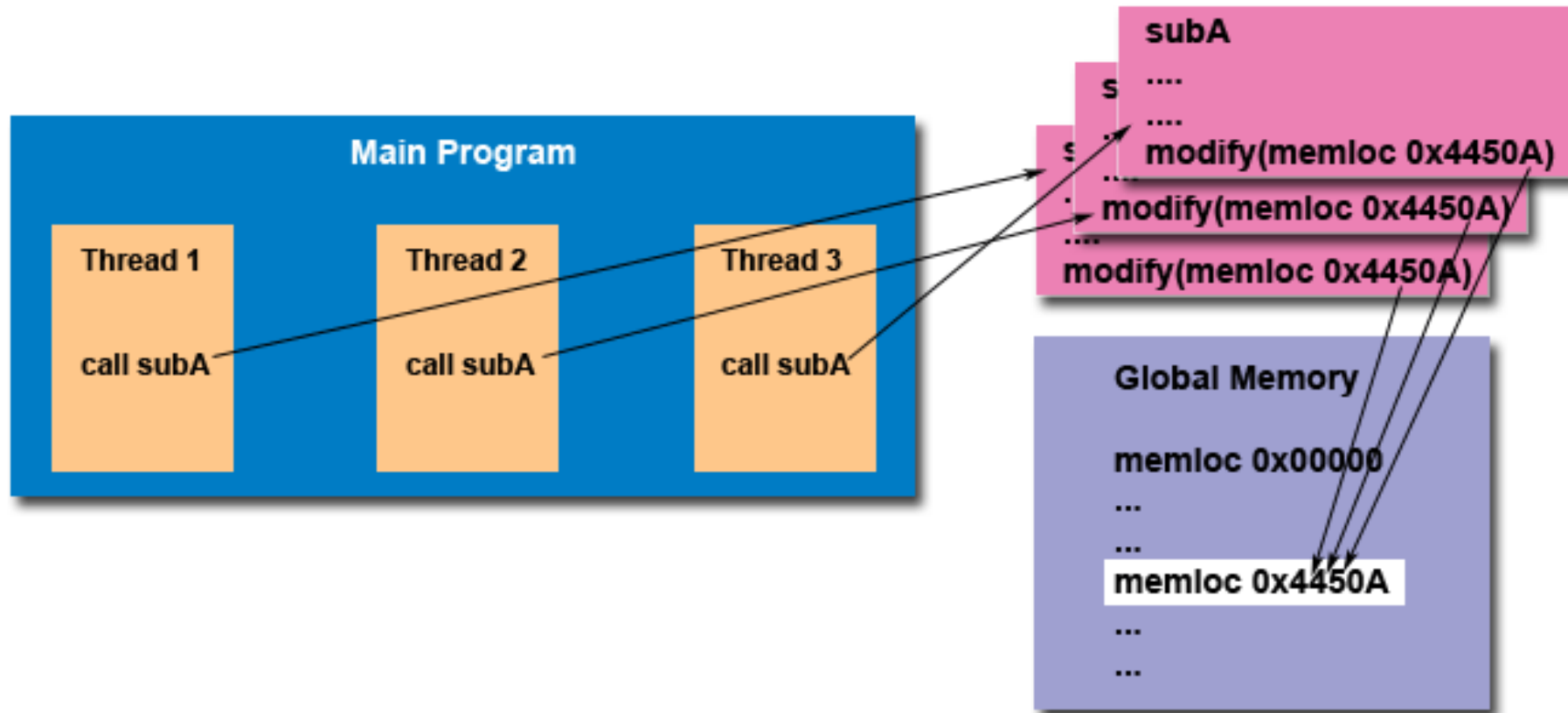


# Using Pthreads, cont

- Programs having the following characteristics may be well suited for pthreads:
- Work that can be executed, or data that can be operated on, by multiple tasks simultaneously:
- Block for potentially long I/O waits
- Use many CPU cycles in some places but not others
- Must respond to asynchronous events
- Some work is more important than other work (priority interrupts)

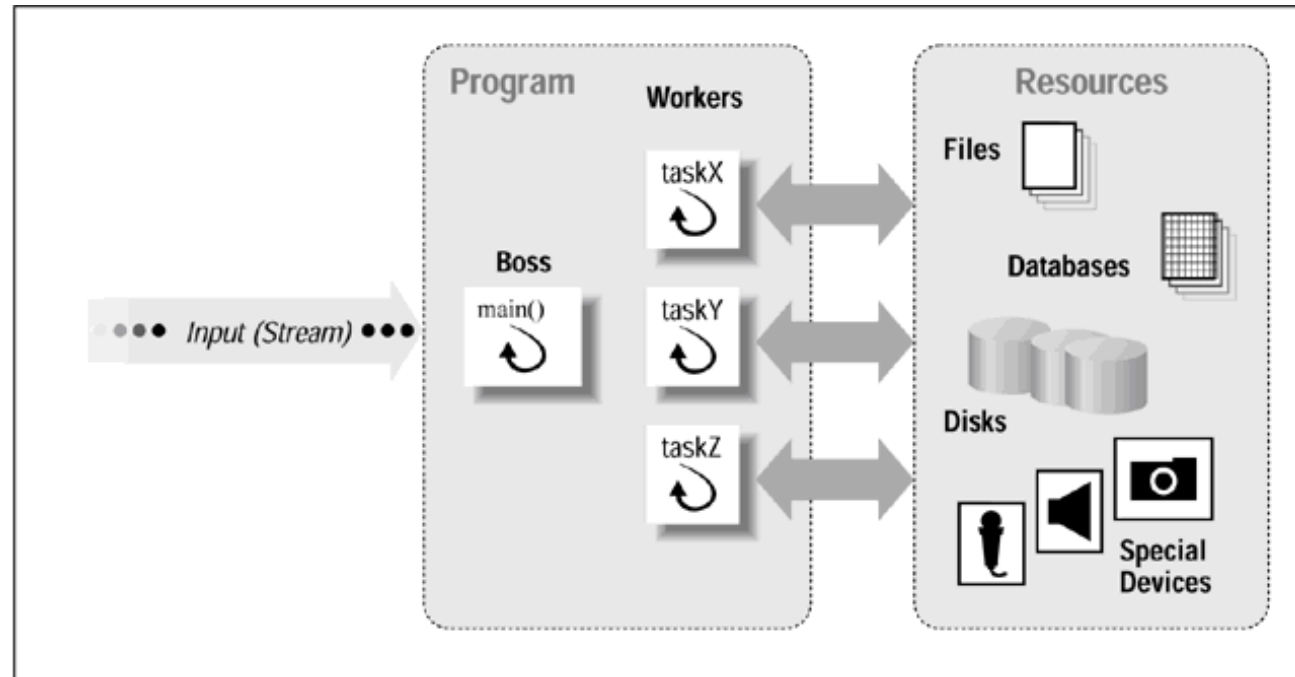


# Thread-Safeness



- Caution: Watch out for executing multiple Pthreads simultaneously that may overwrite shared data.
- Use code to employ synchronization constructs to prevent data corruption

# Manager / Worker Model



- *A single thread, the manager assigns work to other threads, the workers. Typically, the manager handles all input and parcels out work to the other tasks. At least two forms of the manager/worker model are common: static worker pool and dynamic worker pool.*

# Manager / Worker Model

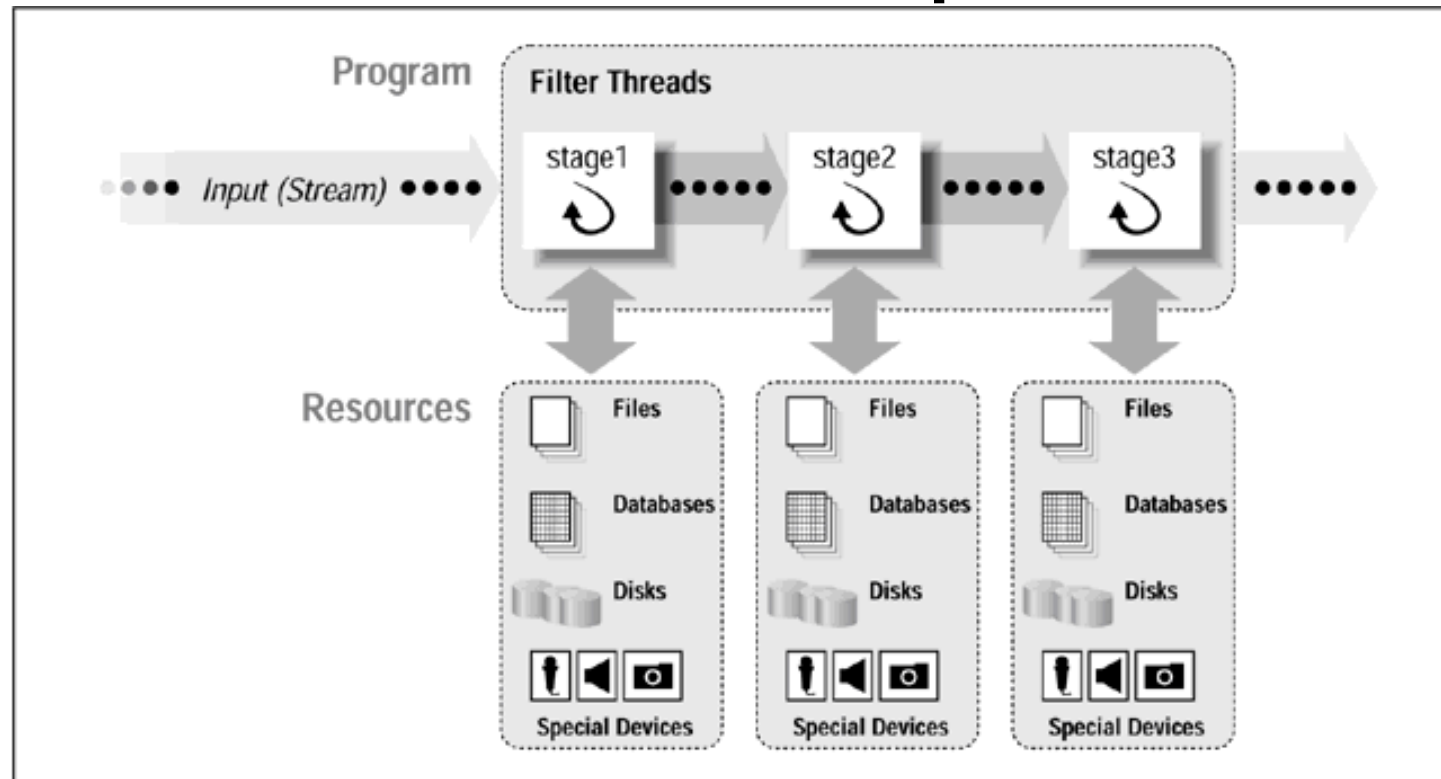
```
main()
/* The Manager */
{
    forever {
        get a request
        switch request
        case X : pthread_create( ... taskX)
        case Y : pthread_create( ... taskY)
    }
}
```

Manager  
delegates  
the task to  
pThreads

```
taskX() /* Workers processing requests of type X */
{
    perform the task, synchronize as needed if accessing shared
    resources
    done
}
taskY() /* Workers processing requests of type Y */
{
    perform the task, synchronize as needed if accessing shared
    resources
    done
}
```

Worker  
pThreads

# Pipeline Model



- *A task is broken into a series of sub-operations, each of which is handled in series, but concurrently, by a different thread. An automobile assembly line best describes this model.*

```
main()
{
    pthread_create( ... stage1 )
    pthread_create( ... stage2 )
    .
    .
    .
    wait for all pipeline threads to finish
    do any clean up
}

stage1()
{
    forever {
        get next input for the program
        do stage 1 processing of the input
        pass result to next thread in pipeline
    }
}

stage2()
{
    forever {
        get input from previous thread in pipeline
        do stage 2 processing of the input
        pass result to next thread in pipeline
    }
}

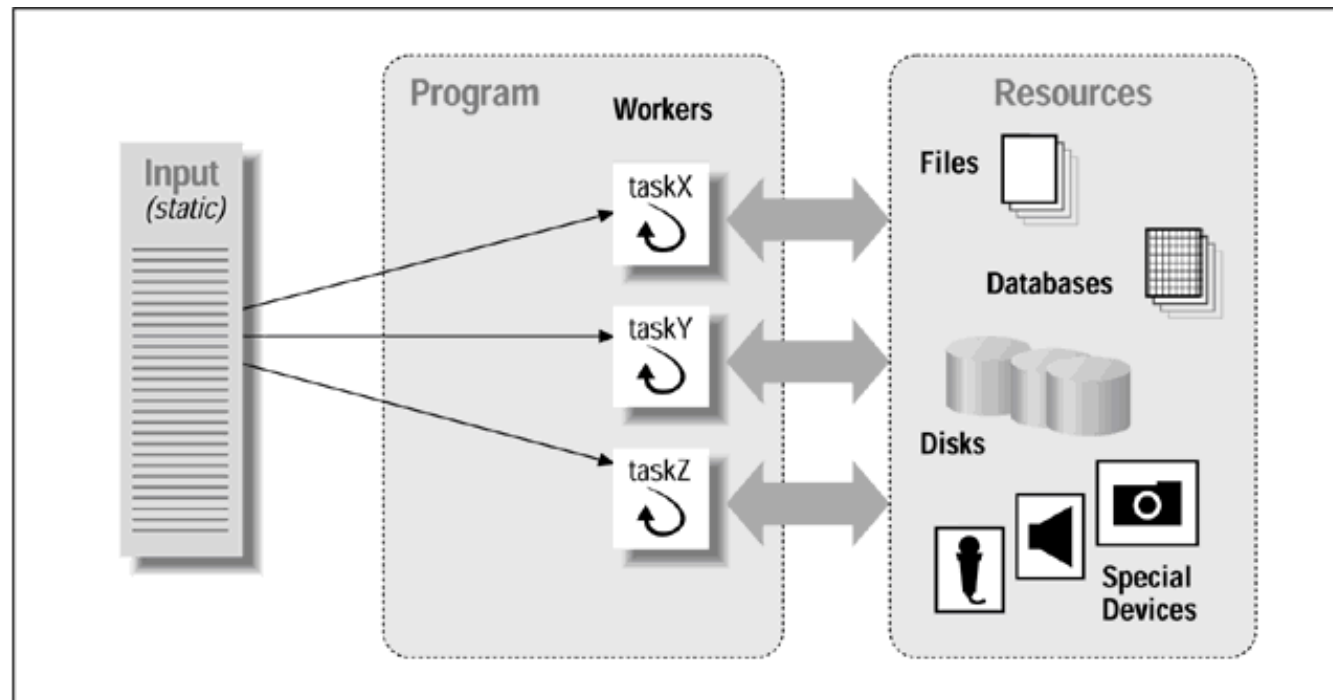
stageN()
{
    forever {
        get input from previous thread in pipeline
        do stage N processing to the input
        pass result to program output
    }
}
```

# Pipeline Model

Manager  
breaks task into  
smaller tasks  
for pThreads

Worker  
pThreads

# Peer Models



- *Similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.*

# Peer Models

```
main()
```

```
{
```

```
    pthread_create( ... thread1 ... task1 )
```

```
    pthread_create( ... thread2 ... task2 )
```

```
    signal all workers to start
```

```
    wait for all workers to finish
```

```
    do any clean up
```

```
}
```

```
task1()
```

```
{
```

```
    wait for start
```

```
    perform task, synchronize as needed if accessing shared  
resources
```

```
    done
```

```
}
```

```
task2()
```

```
{
```

```
    wait for start
```

```
    perform task, synchronize as needed if accessing shared  
resources
```

```
    done
```

```
}
```

**Manager  
delegates  
the task to  
pThreads**

**Manager  
pThread  
joins worker  
pThreads**