Comp 206
Introduction to Operating Systems

Threads & Concurrency

Dr. Gikaru
Computer Science
Egerton University

Threads

- **Processes have the following components:**
  - an address space
  - a collection of operating system state
  - a CPU context … or *thread* of control

- **On multiprocessor systems, with several CPUs, it would make sense for a process to have several CPU contexts (threads of control)**
  - Thread fork creates new thread not memory space
  - Multiple threads of control could run in the same memory space on a single CPU system too!

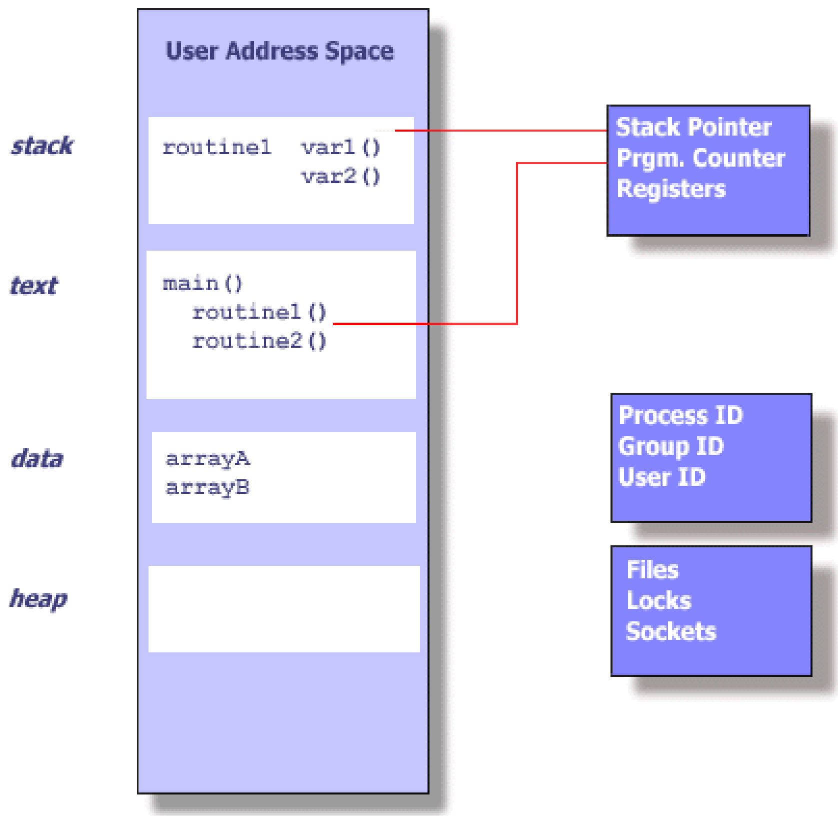
Threads

- **Threads share a process address space with zero or more other threads**

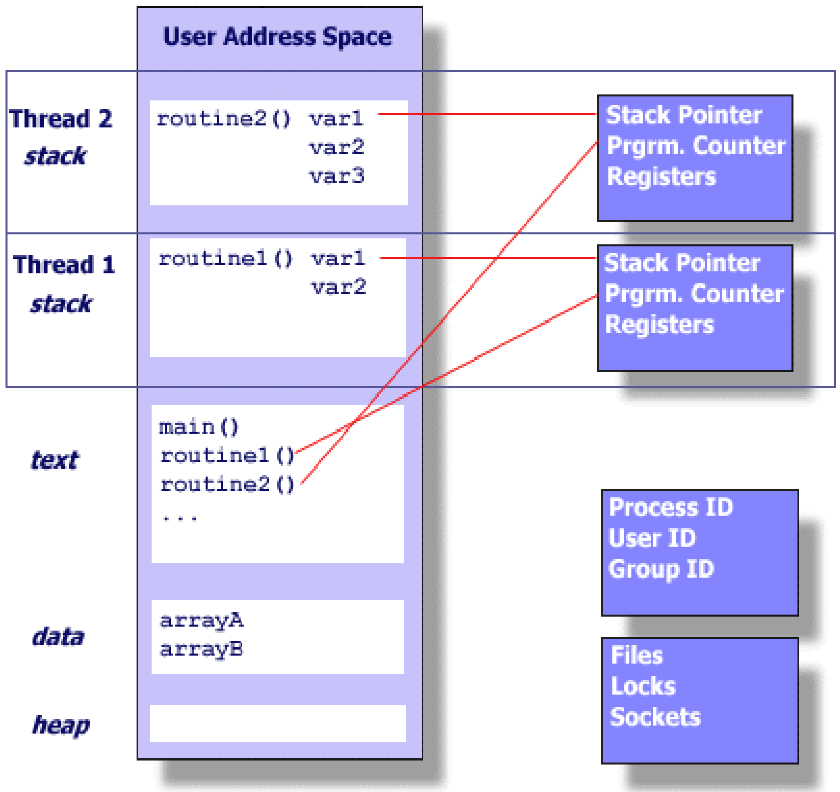
- **Threads have their own**
  - PC, SP, register state, stack

- **A traditional process can be viewed as a memory address space with a single thread**

Single thread state within a process



Multiple threads in an address space



What is a thread?

- **A thread executes a stream of instructions**
  - it is an abstraction for control-flow
- **Practically, it is a processor context and stack**
  - Allocated a CPU by a scheduler
  - Executes in the context of a memory address space

Summary of private per-thread state

Things that define the state of a particular flow of control in an executing program:

- Stack (local variables)
- Stack pointer
- Registers
- Scheduling properties (i.e., priority)

Shared state among threads

Things that relate to an instance of an executing program (that may have multiple threads)

- User ID, group ID, process ID
- Address space
  - **Text**
  - **Data (off-stack global variables)**
  - **Heap (dynamic data)**
- Open files, sockets, locks

Important: Changes made to shared state by one thread will be visible to the others

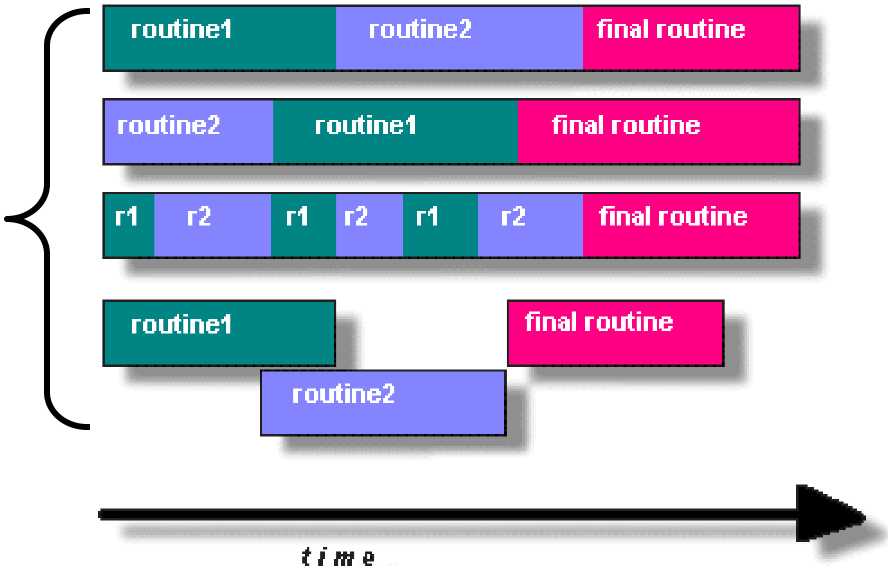
- Reading and writing memory locations requires synchronization! … a major topic for later …

How do you program using threads?

Split program into routines to execute in parallel

- True or pseudo (interleaved) parallelism

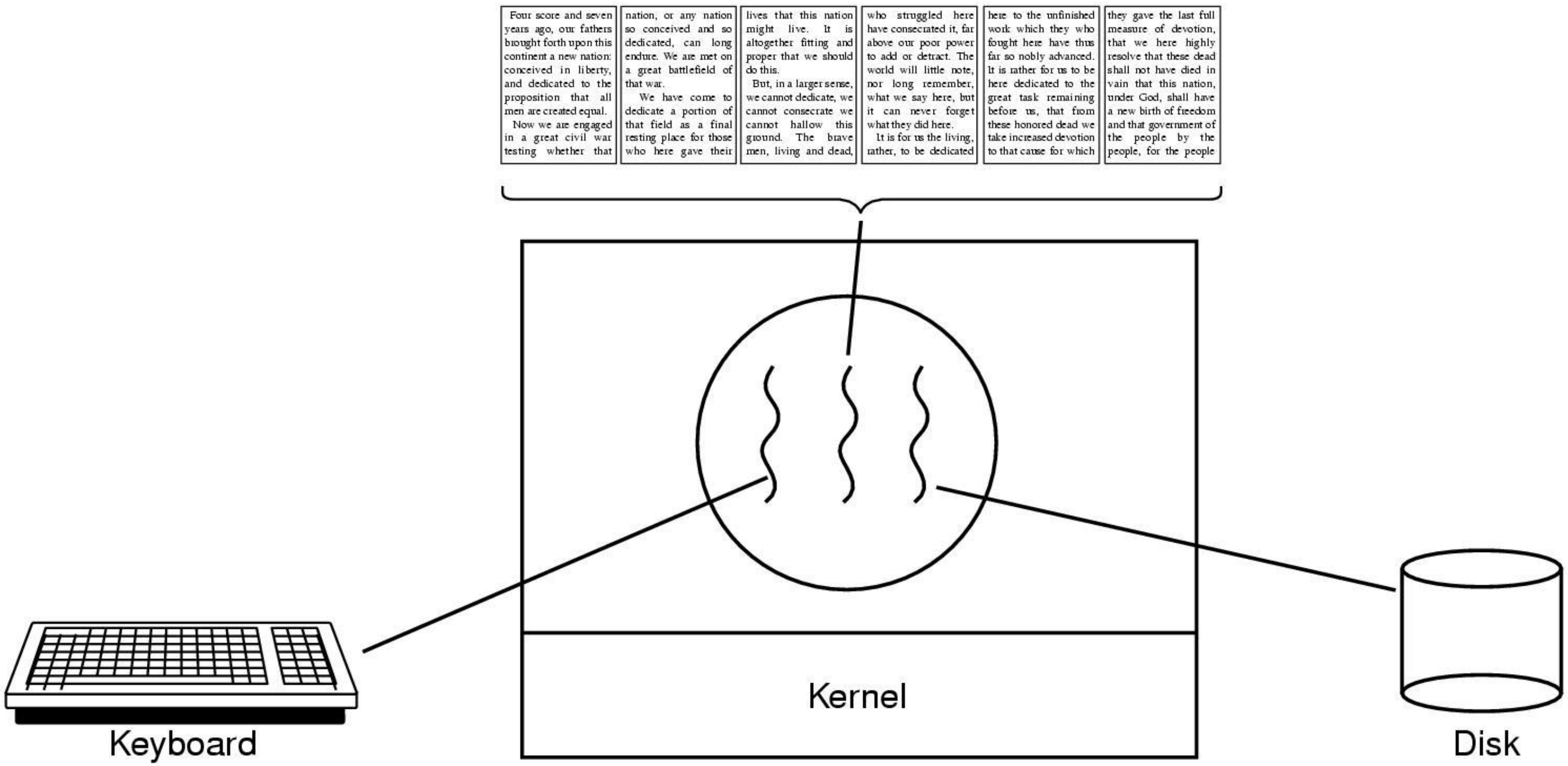
Alternative strategies for executing multiple rountines



Why program using threads?

- **Utilize multiple CPU's concurrently**
- **Low cost communication via shared memory**
- **Overlap computation and blocking on a single CPU**
  - Blocking due to I/O
  - Computation and communication
- **Handle asynchronous events**

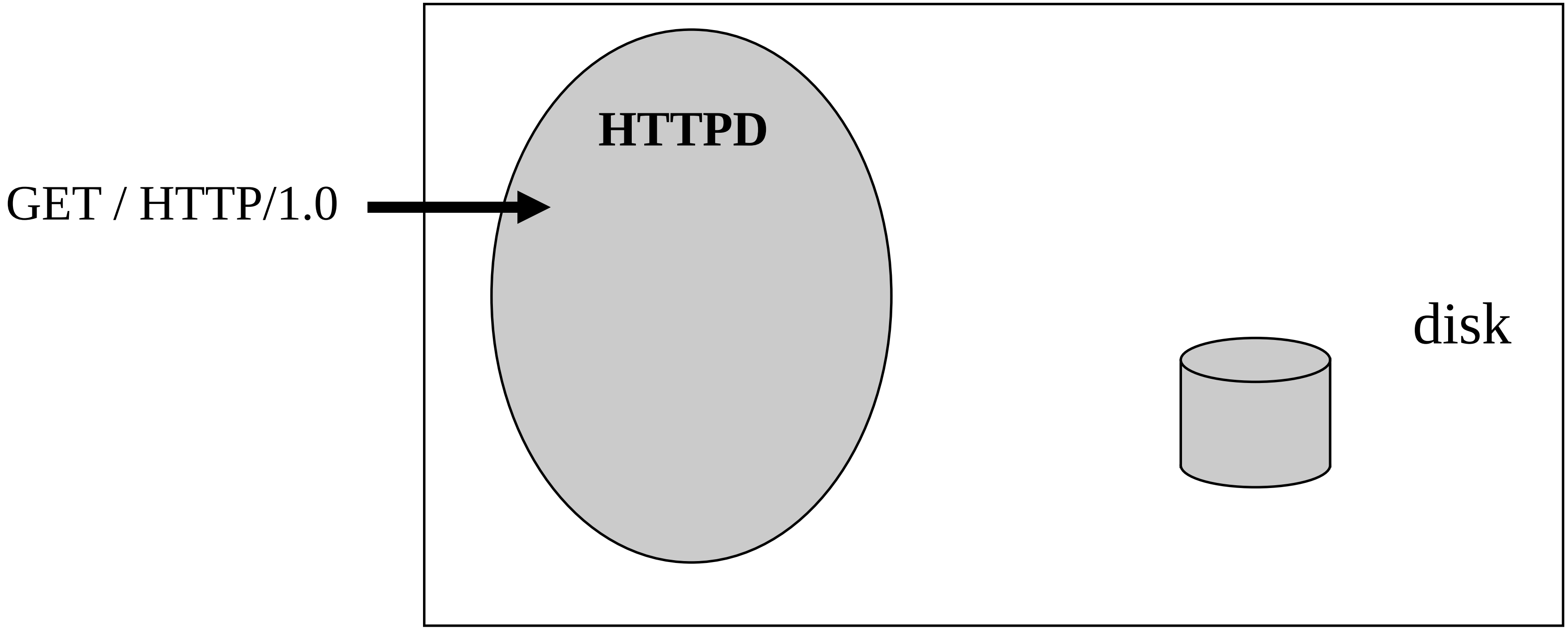
Thread usage



A word processor with three threads

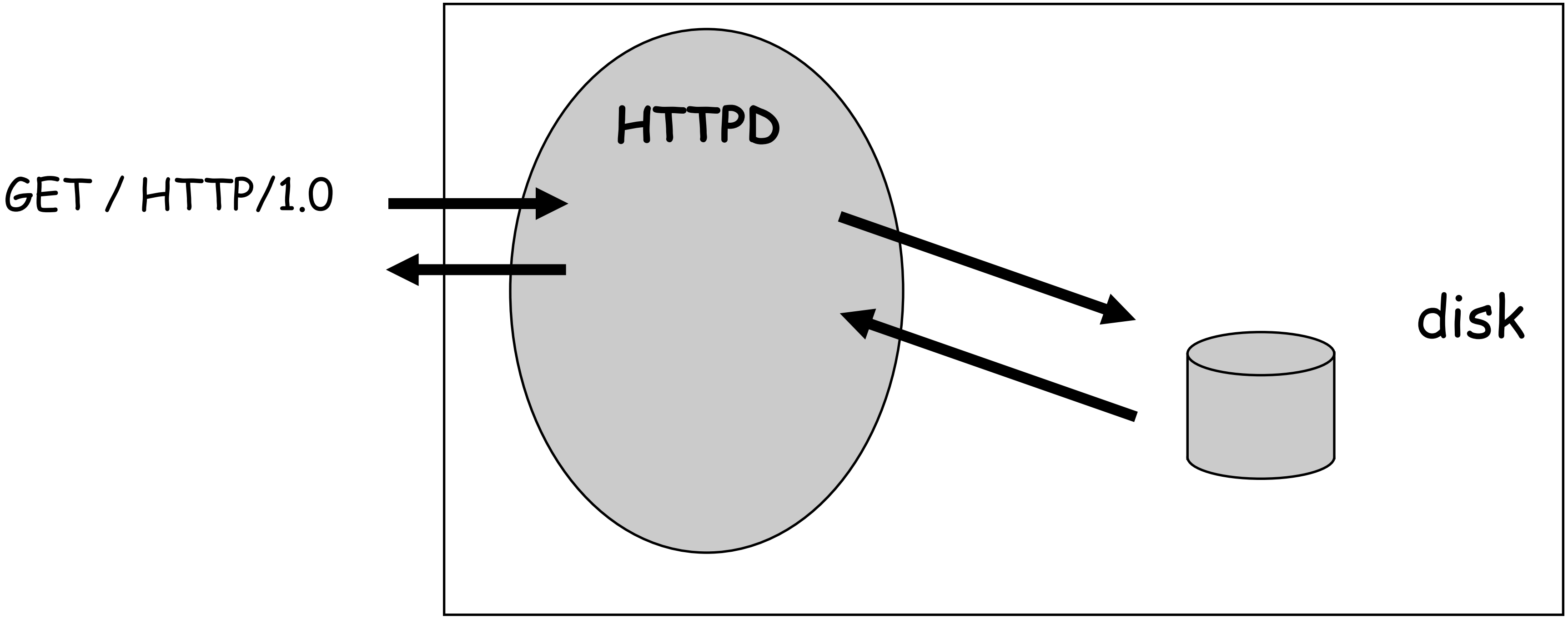
Processes versus threads - example

- **A WWW process**



Processes versus threads - example

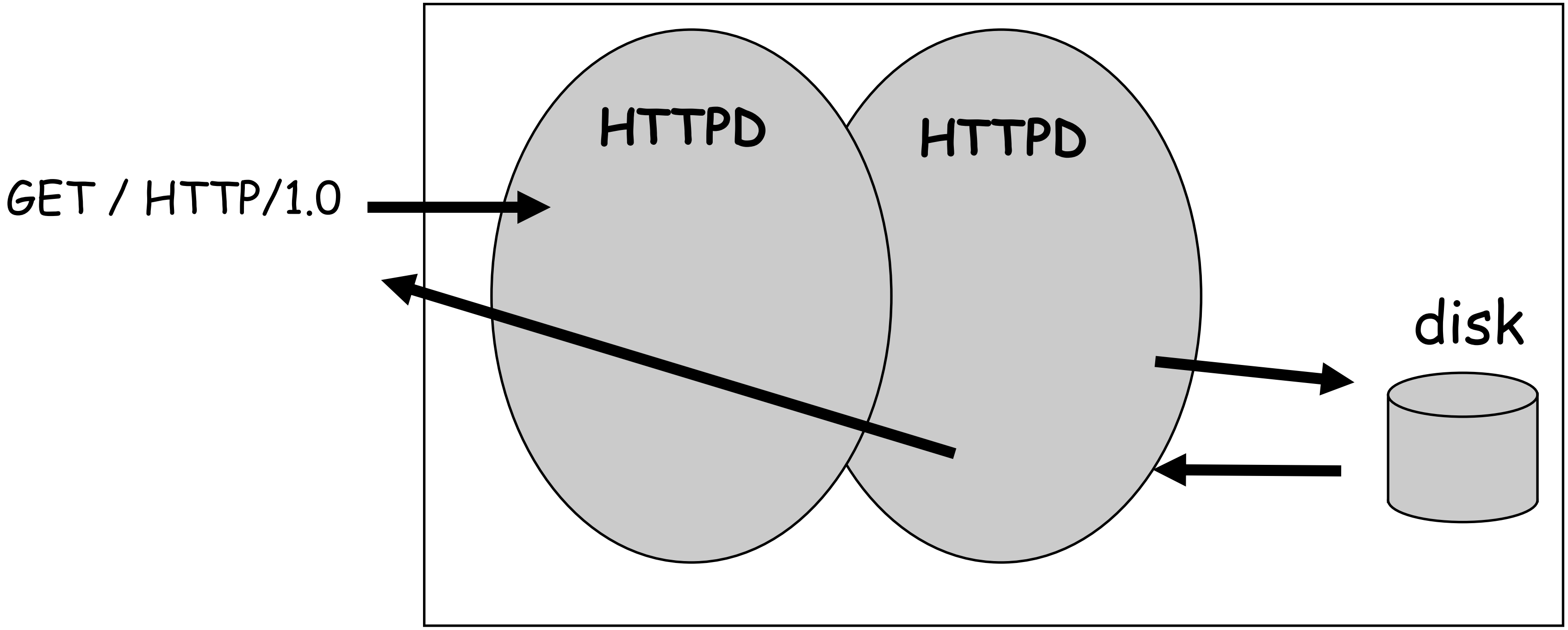
- **A WWW process**



Why is this not a good web server design?

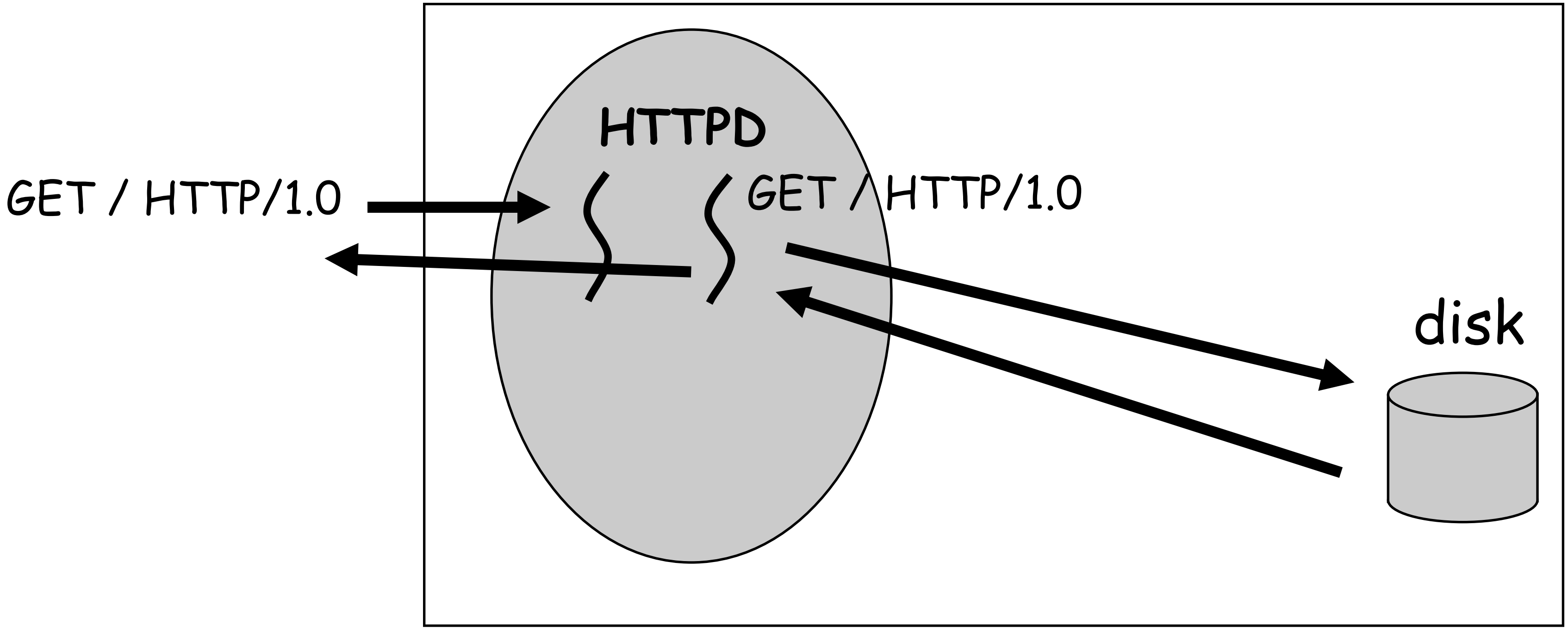
Processes versus threads - example

- **A WWW process**



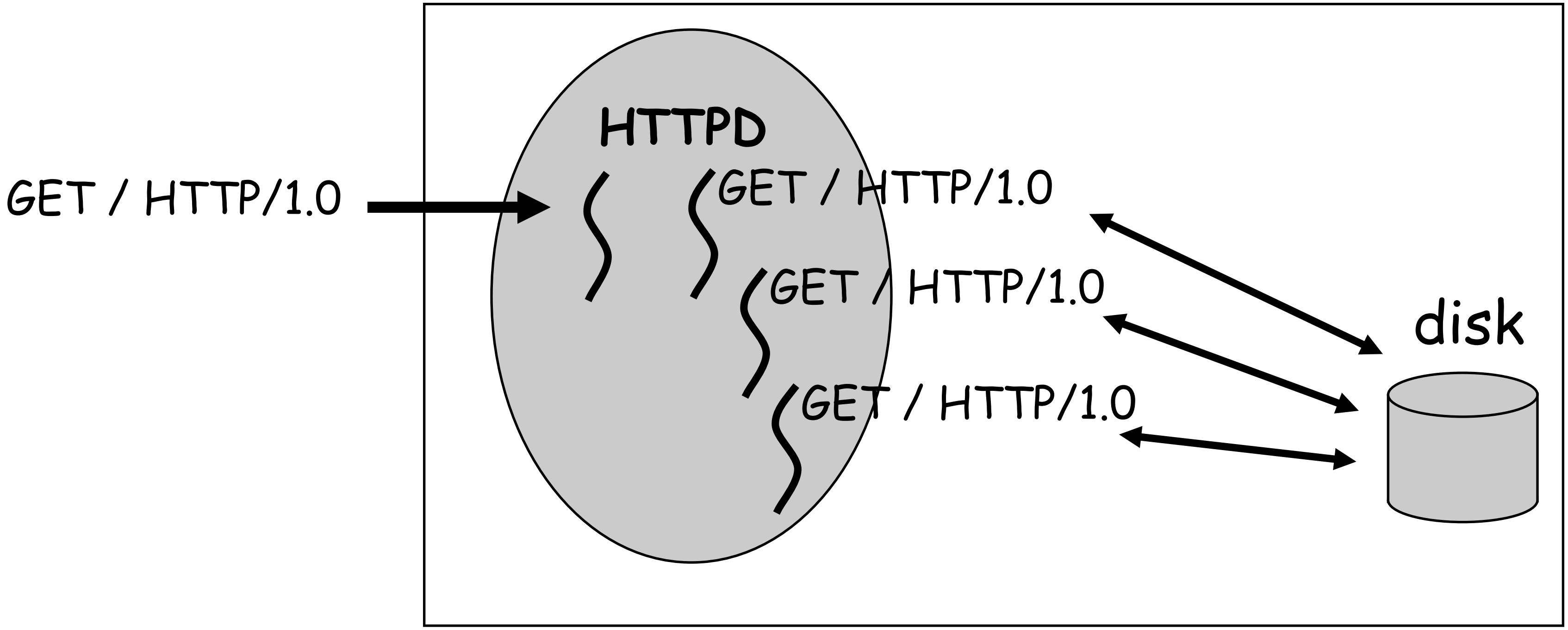
Processes versus threads - example

- A WWW process

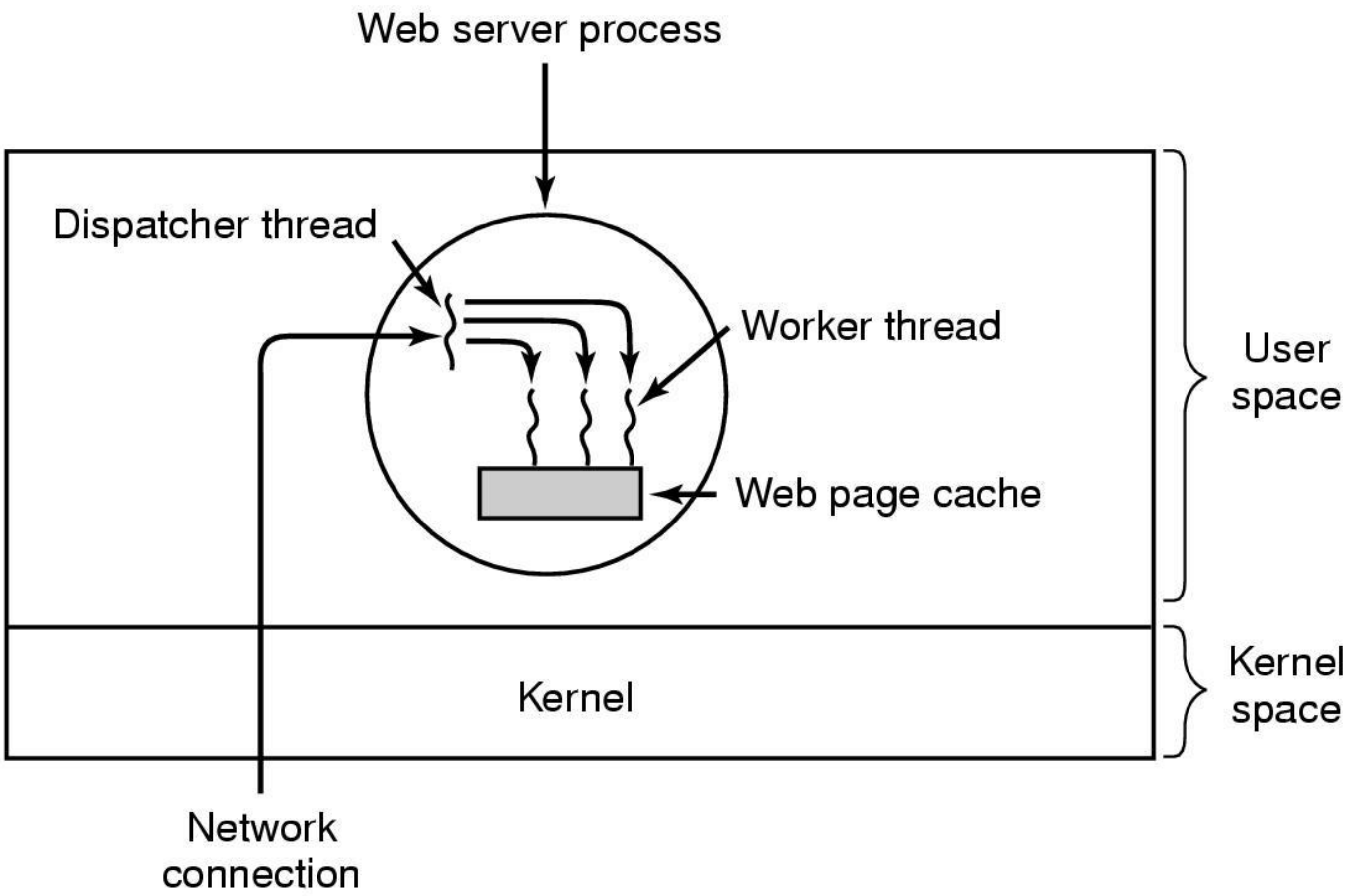


Processes versus threads - example

- A WWW process



Threads in a web server



A multithreaded web server

Thread usage

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
       read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

- **Rough outline of code for previous slide**
  - (a) Dispatcher thread
  - (b) Worker thread

System structuring options

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Three ways to construct a server

Common thread programming models

- **Manager/worker**
  - Manager thread handles I/O and assigns work to worker threads
  - Worker threads may be created dynamically, or allocated from a thread-pool

- **Pipeline**
  - Each thread handles a different stage of an assembly line
  - Threads hand work off to each other in a producer-consumer relationship

What does a typical thread API look like?

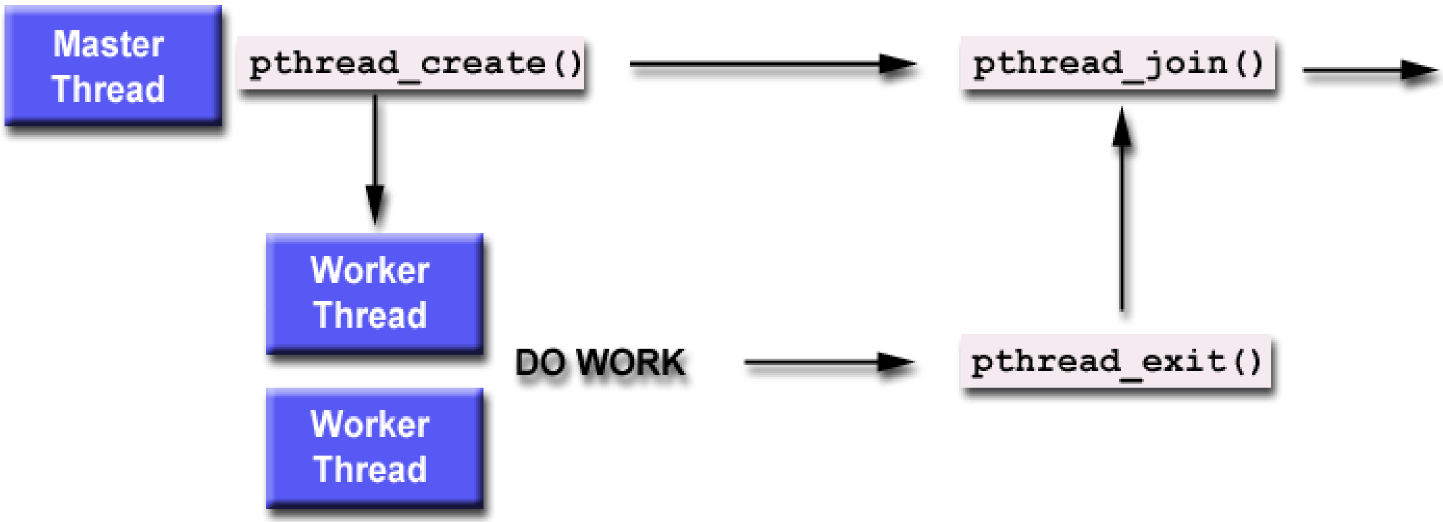
- **POSIX standard threads (Pthreads)**
- **First thread exists in main(), typically creates the others**

- **pthread_create (thread,attr,start_routine,arg)**
  - Returns new thread ID in "thread"
  - Executes routine specified by "start_routine" with argument specified by "arg"
  - Exits on return from routine or when told explicitly

Thread API (continued)

- **pthread_exit (status)**
  - Terminates the thread and returns "status" to any joining thread
- **pthread_join (threadid,status)**
  - Blocks the calling thread until thread specified by "threadid" terminates
  - Return status from pthread_exit is passed in "status"
  - One way of synchronizing between threads
- **pthread_yield ()**
  - Thread gives up the CPU and enters the run queue

Using create, join and exit primitives



An example Pthreads program

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
  printf("\n%d: Hello World!\n", threadid);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc, t;
  for(t=0; t<NUM_THREADS; t++)
  {
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
    if (rc)
    {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

Program Output

```
Creating thread 0
Creating thread 1
0: Hello World!
1: Hello World!
Creating thread 2
Creating thread 3
2: Hello World!
3: Hello World!
Creating thread 4
4: Hello World!
```

For more examples see: http://www.llnl.gov/computing/tutorials/pthreads

Pros & cons of threads

- **Pros**
  - Overlap I/O with computation!
  - Cheaper context switches
  - Better mapping to shared memory multiprocessors
- **Cons**
  - Potential thread interactions
  - Complexity of debugging
  - Complexity of multi-threaded programming
  - Backwards compatibility with existing code

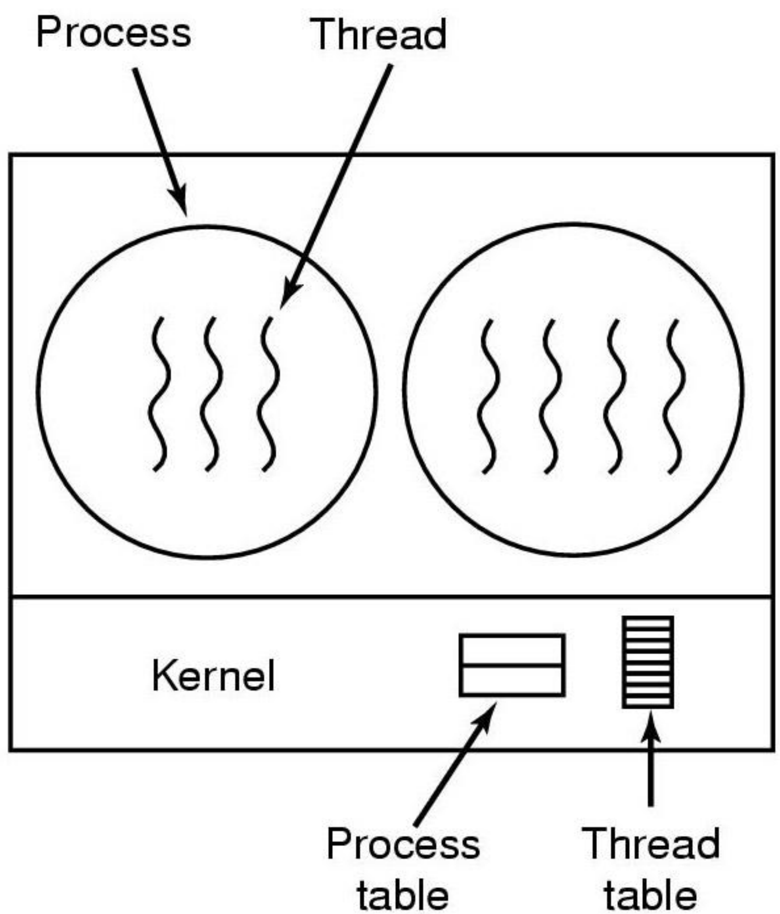
User-level threads

- **The idea of managing multiple abstract program counters above a single real one can be implemented using privileged or non-privileged code.**
  - Threads can be implemented in the OS or at user level

- **User level thread implementations**
  - thread scheduler runs as user code (thread library)
  - manages thread contexts in user space
  - The underlying OS sees only a traditional process above

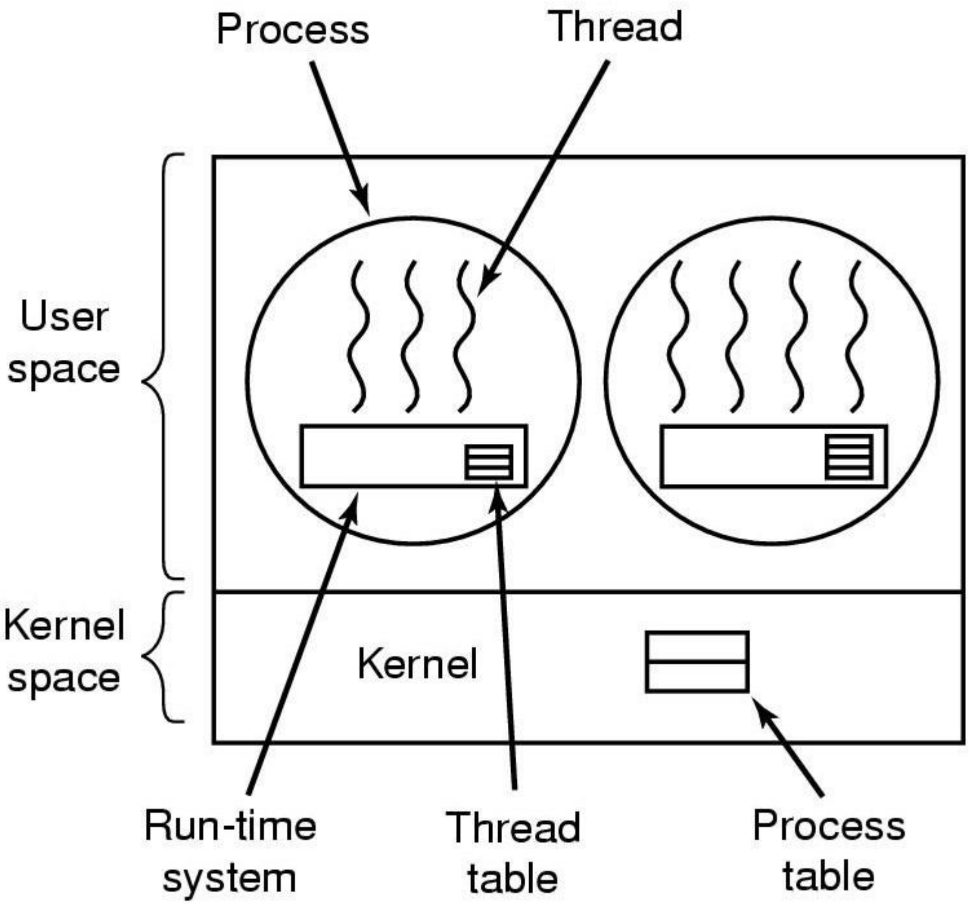
Kernel-level threads

The thread-switching code is in the kernel



User-level threads package

The thread-switching code is in user space



User-level threads

- **Advantages**
  - cheap context switch costs among threads in the same process!
    - **A procedure call not a system call!**
  - User-programmable scheduling policy
- **Disadvantages**
  - How to deal with blocking system calls!
  - How to overlap I/O and computation!

END