Operating Systems

Lecture 3

Threads & Concurrency

Threads

Processes have the following components:

- an address space
- a collection of operating system state
- a CPU context … or *thread* of control

To use multiple CPUs on a multiprocessor system, a process would need several CPU contexts

- 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

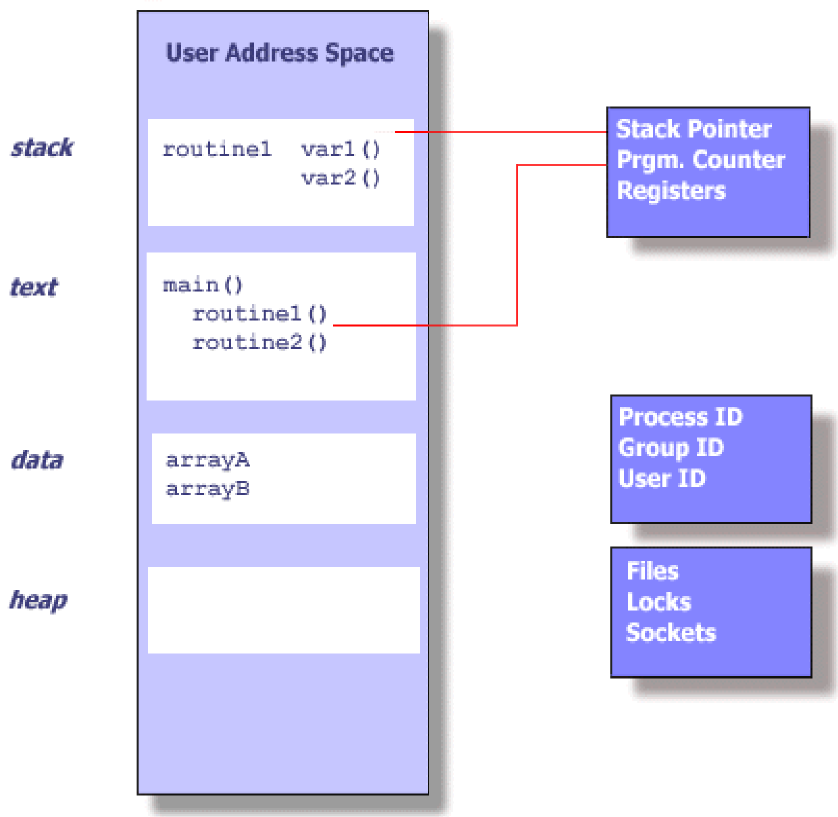
Threads share a process address space with zero or more other threads

Threads have their own CPU context

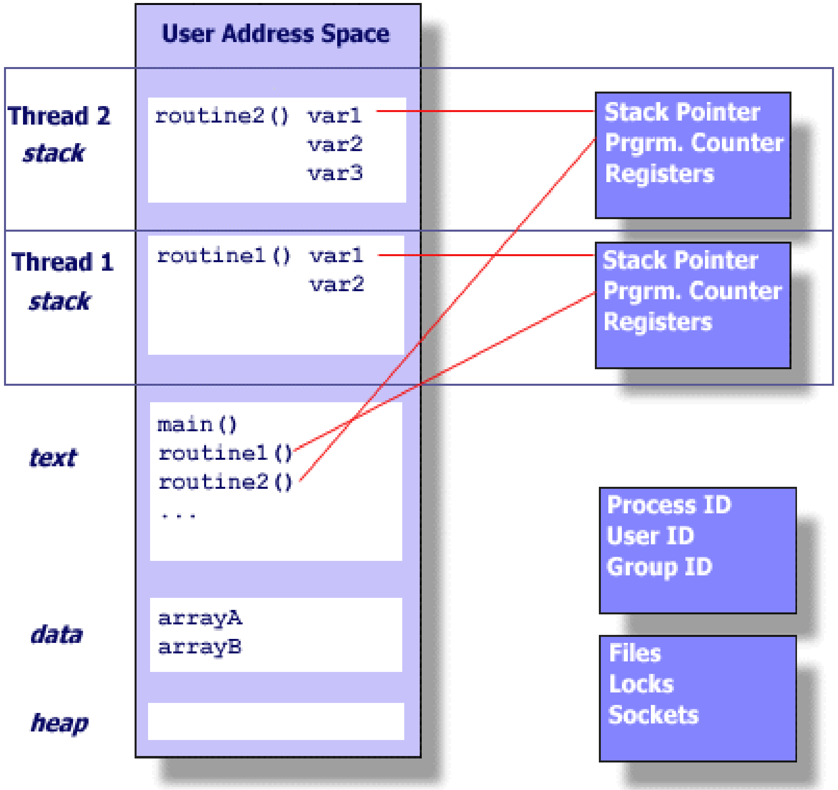
- PC, SP, register state,
- Stack

A traditional process could be viewed as a memory address space with a single thread

Single Thread in Address Space



Multiple Threads in 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 a memory address space

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

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

Concurrent Access to Shared State

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

Reading and writing memory locations requires synchronization!

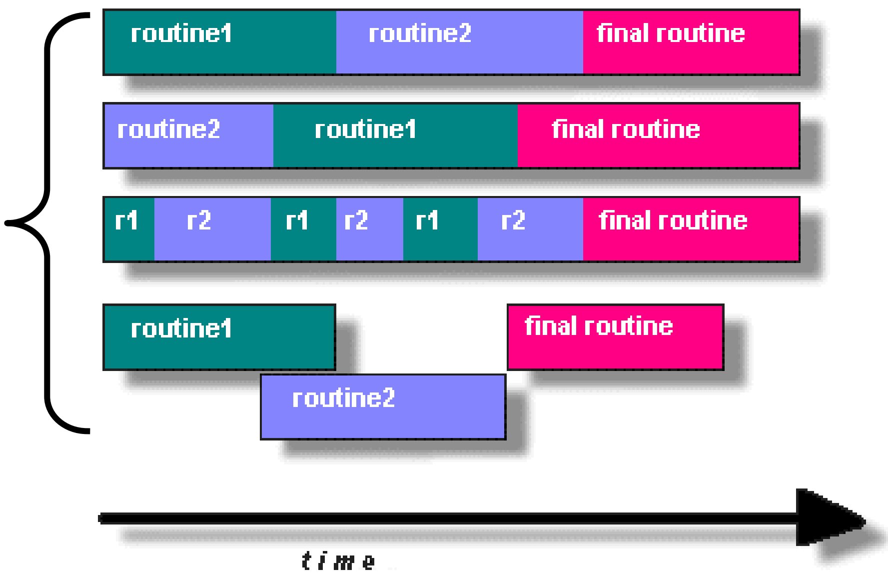
This is a major topic for later …

Programming With Threads

Split program into routines to execute in parallel

- True or pseudo (interleaved) parallelism

Alternative strategies for executing multiple rountines



Why Use Threads?

Utilize multiple CPU's concurrently

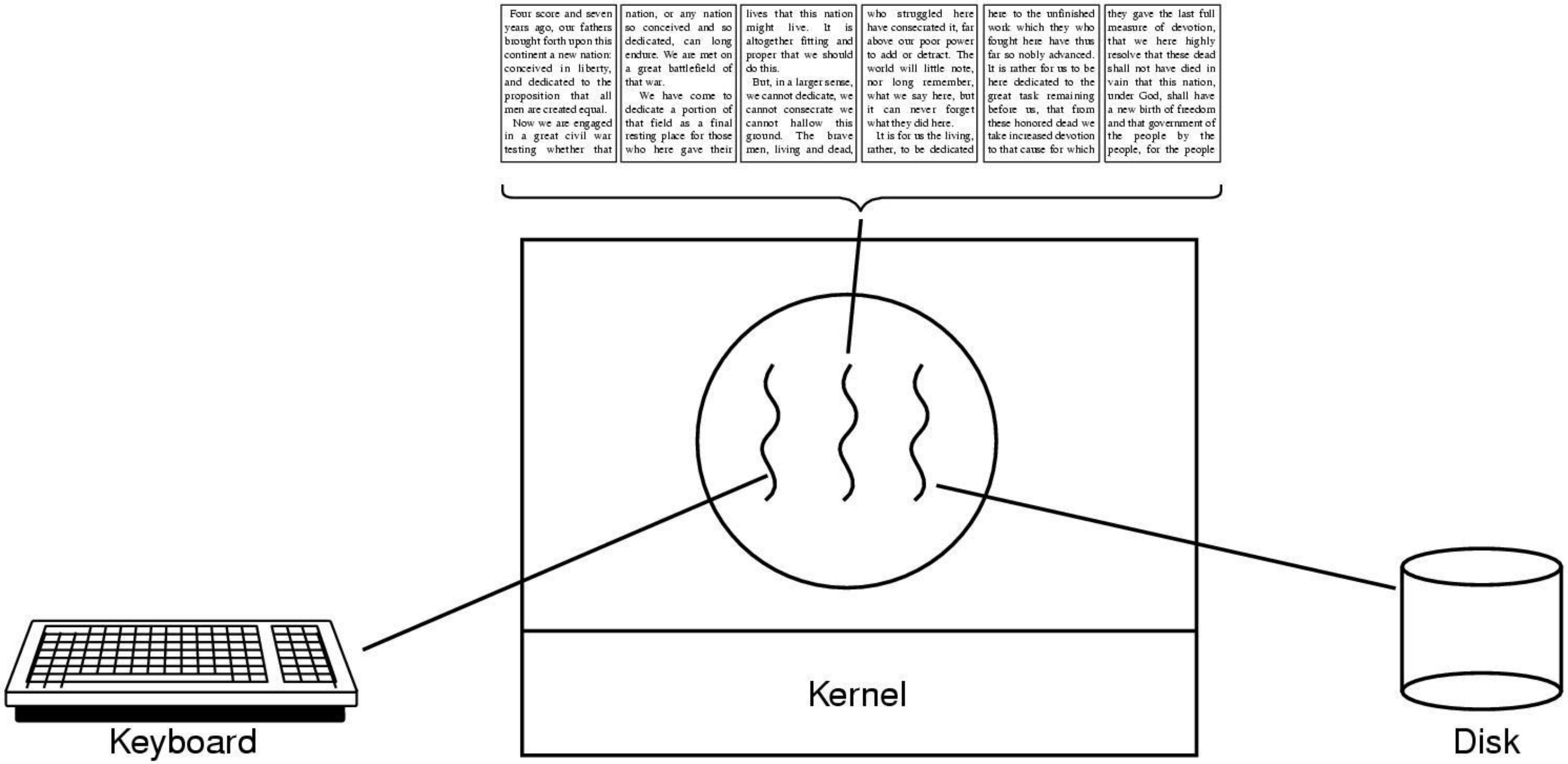
Low cost communication via shared memory

Overlap computation and blocking on a single CPU

- Blocking due to I/O
- Computation and communication

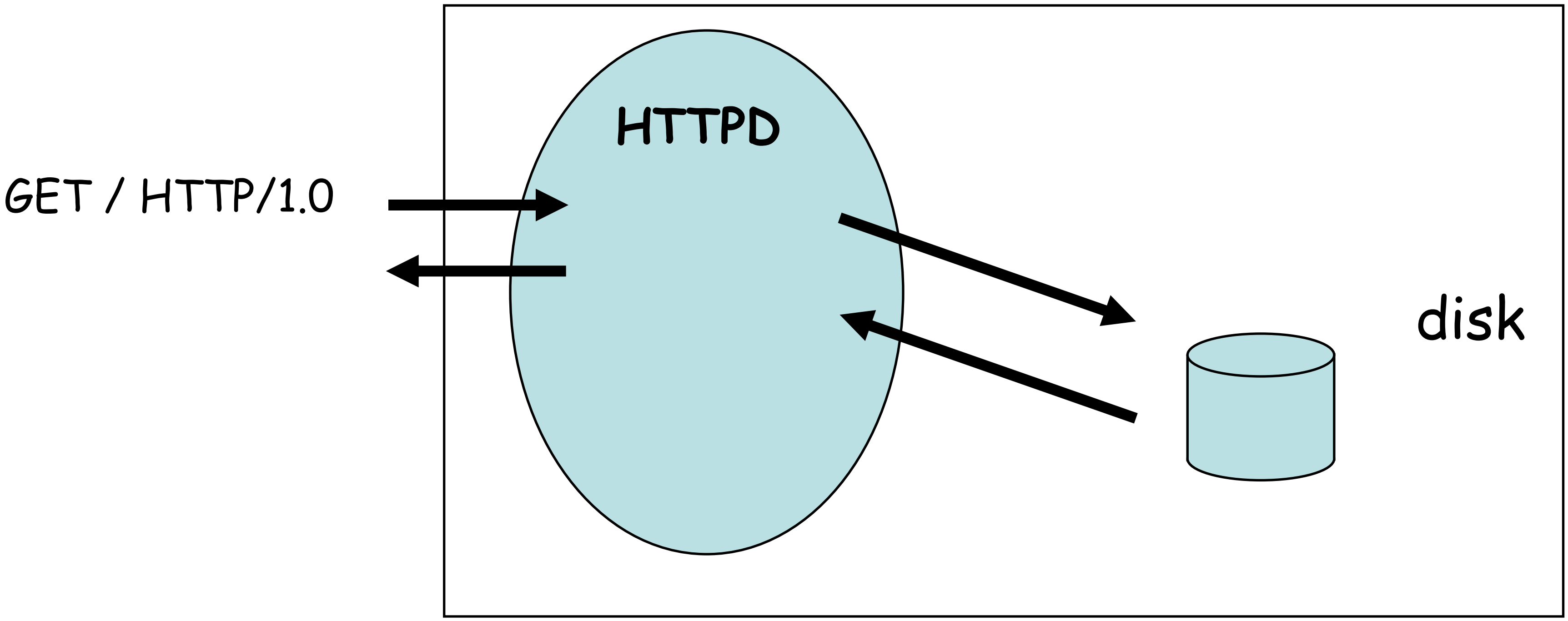
Handle asynchronous events

Typical Thread Usage

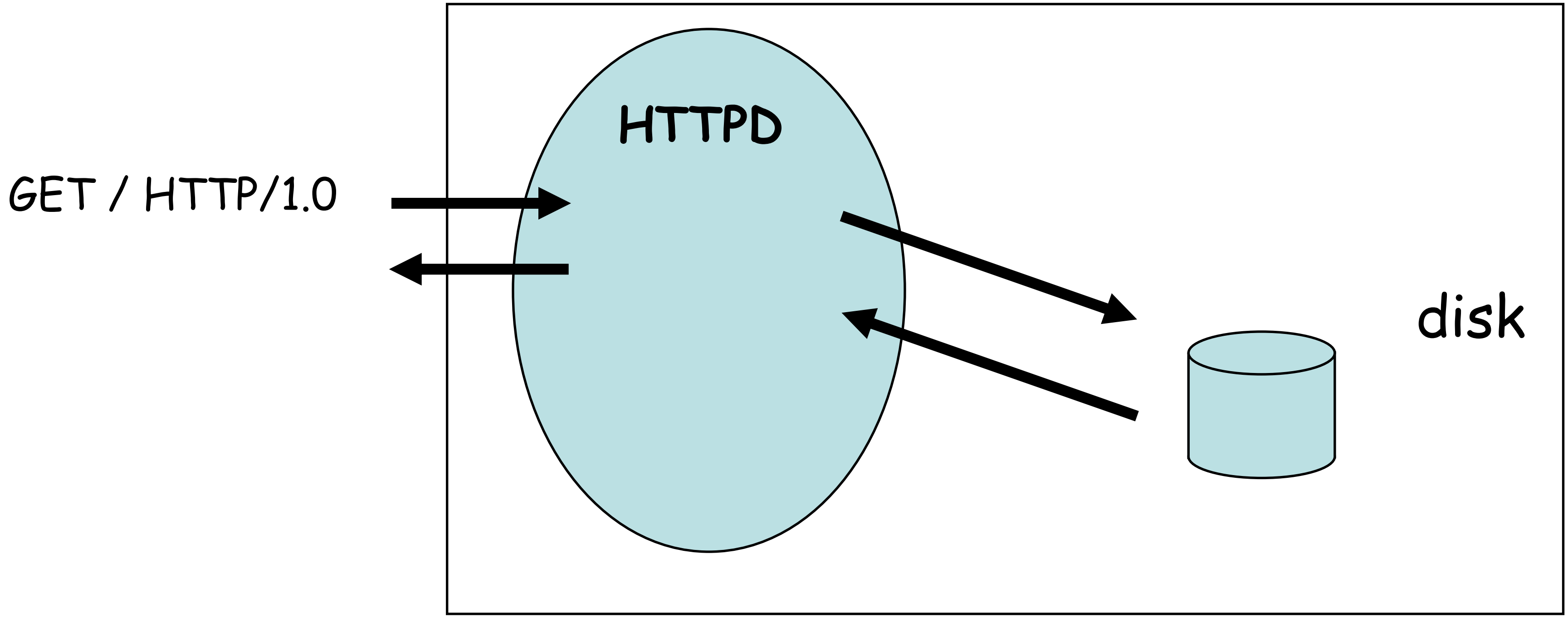


A word processor with three threads

Processes vs Threads

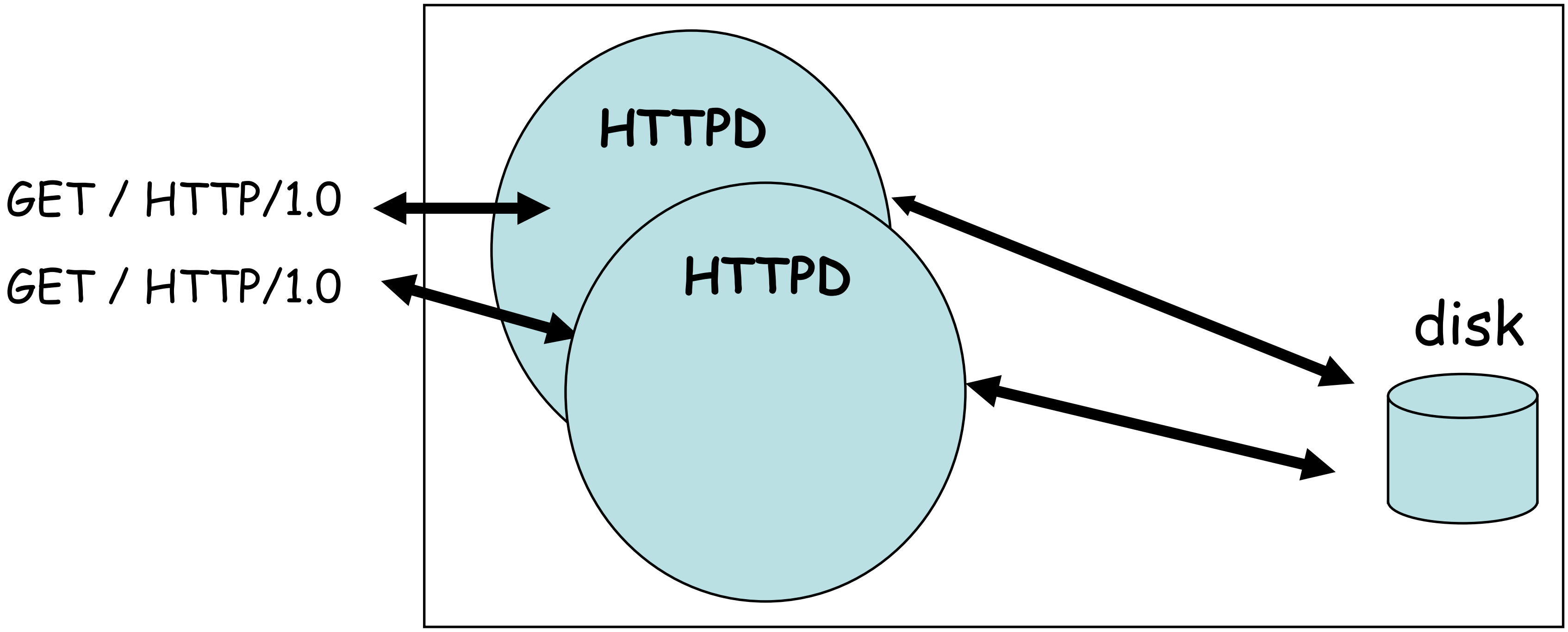


Processes vs Threads

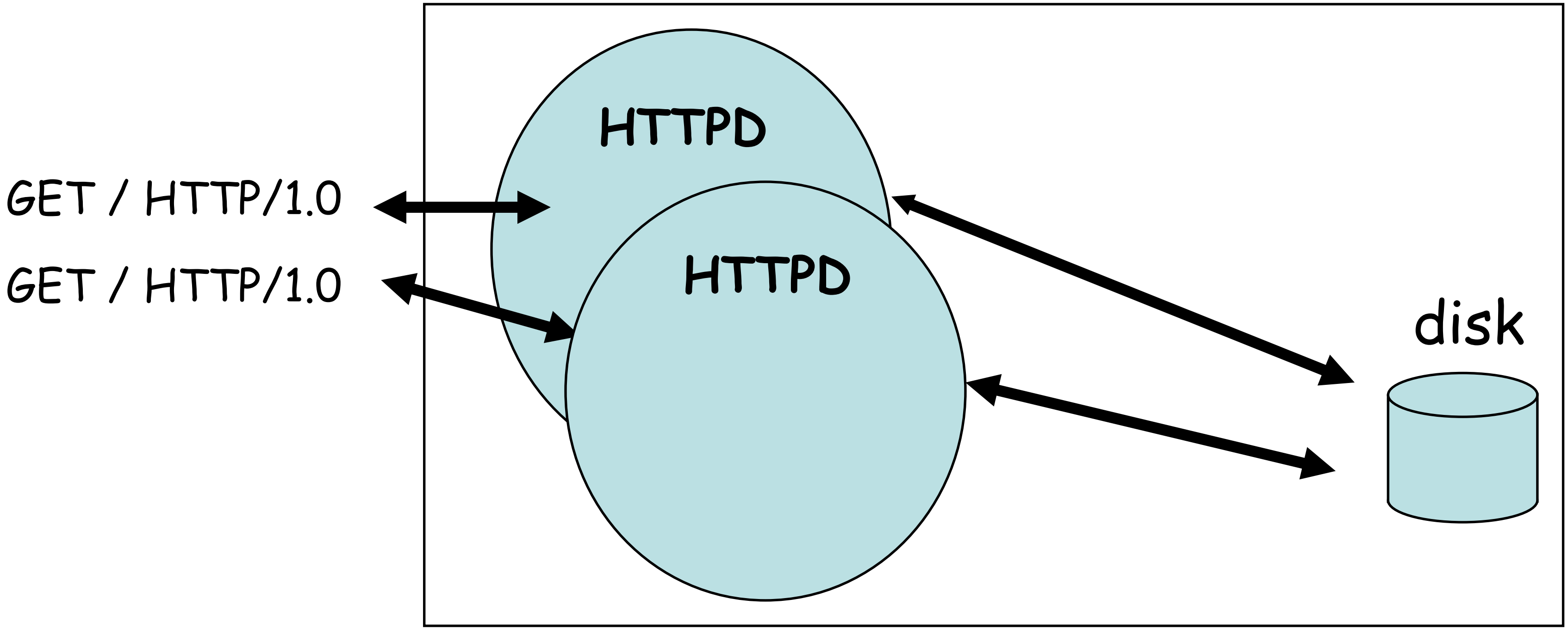


Why is this not a good web server design?

Processes vs Threads

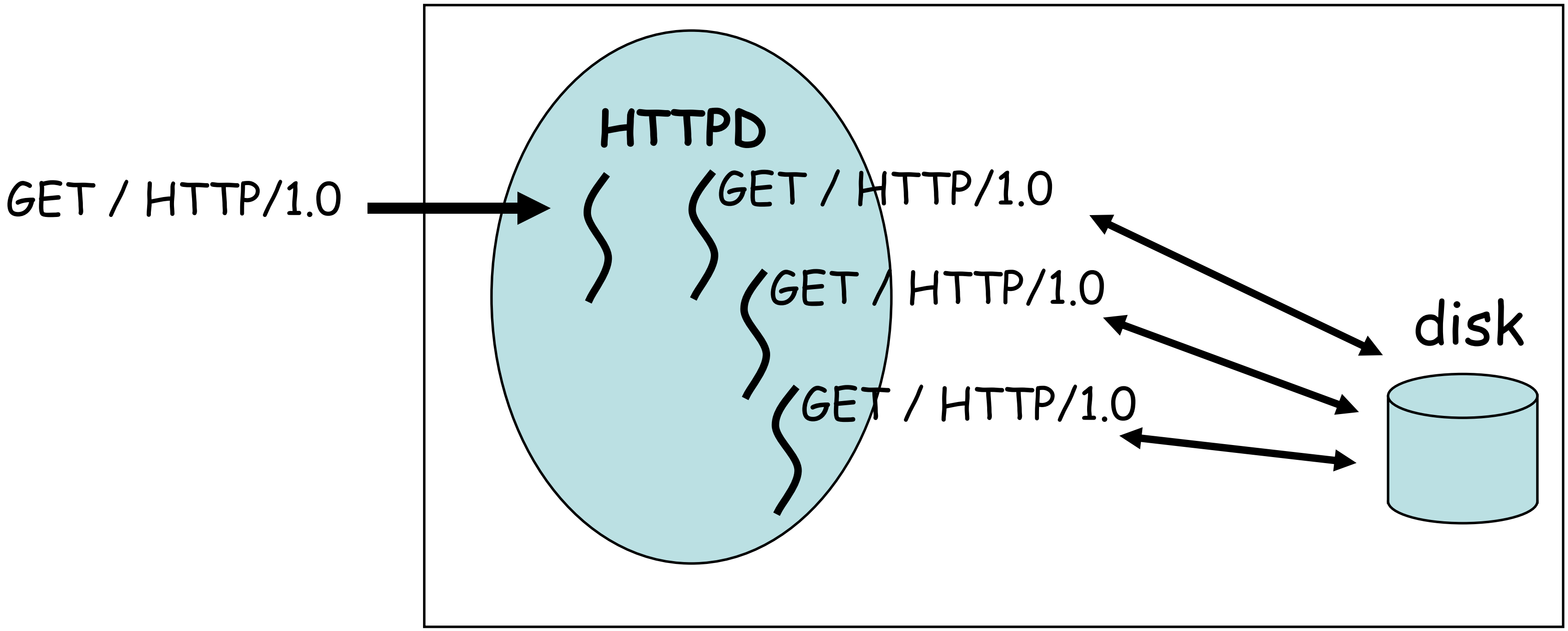


Processes vs Threads



Why is this not a good web server design?

Processes vs Threads



Common Thread Strategies

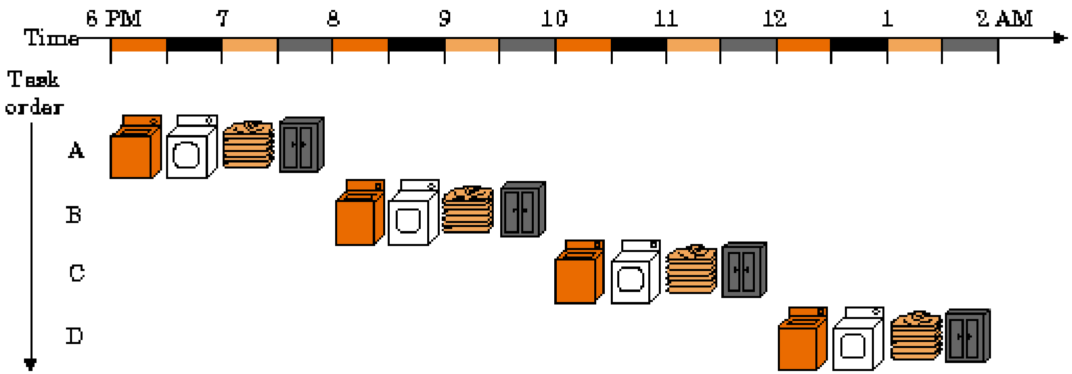
Manager/worker

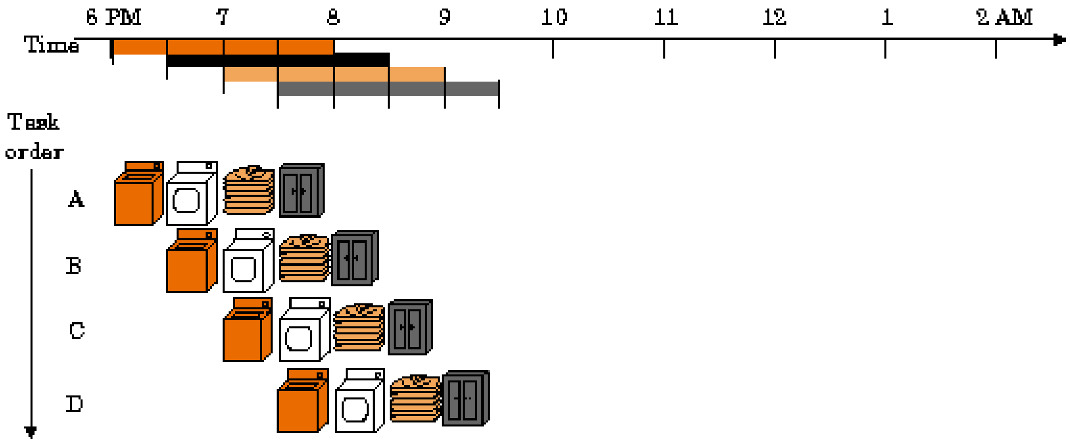
- Manager thread handles I/O
- Manager assigns work to worker threads
- Worker threads created dynamically
- ... or allocated from a *thread-pool*

Pipeline

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

Example of pipeline





Pthreads: A Typical Thread API

Pthreads: POSIX standard threads

First thread exists in main(), 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

Pthreads (continued)

pthread_exit (status)

- Terminates the thread and returns "status" to any joining thread

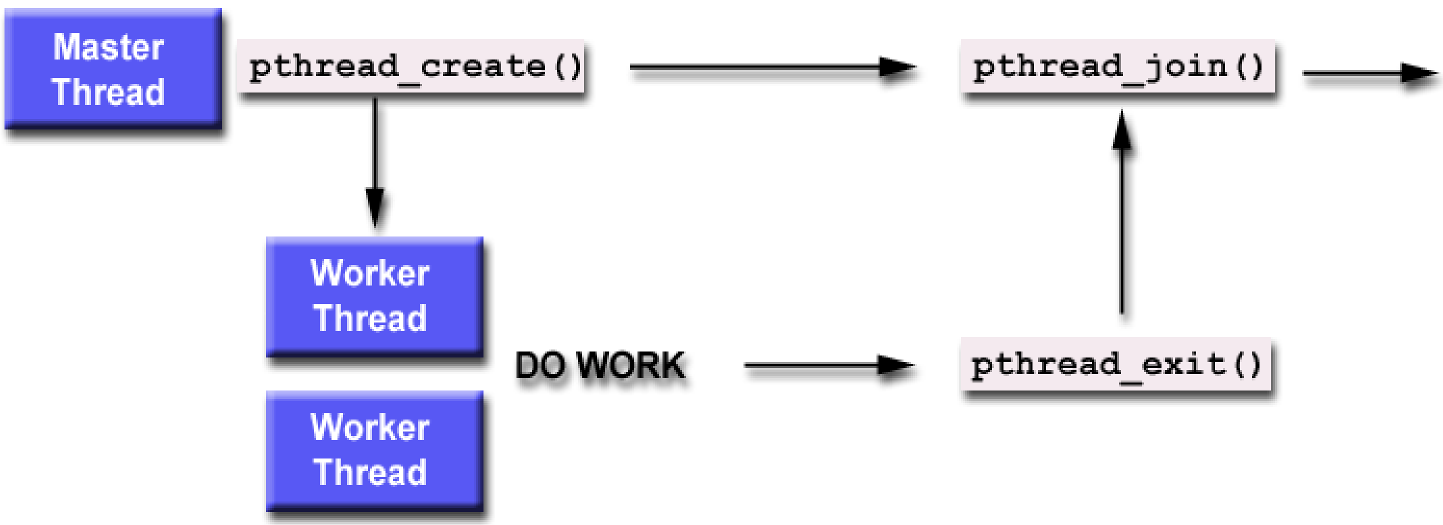
pthread_join (threadid,status)

- 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



An Example Pthreads Program

```c
#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!
```

Pros & Cons of Threads

Pros:

- Overlap I/O with computation!
- Cheaper context switches
- Better mapping to multiprocessors

Cons:

- 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.

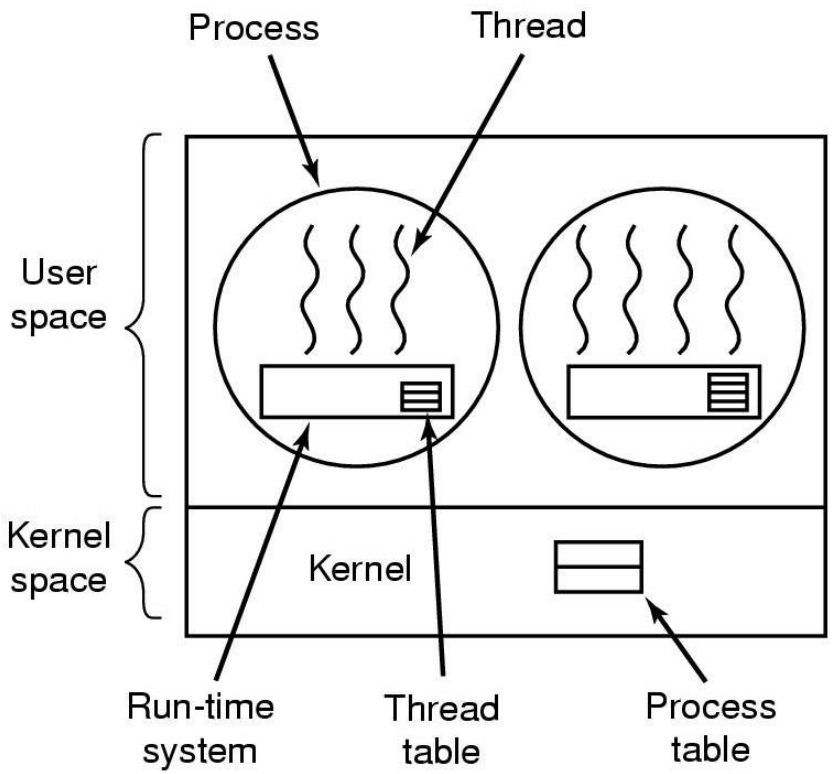
- Threads can be implemented in the OS or at user level

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

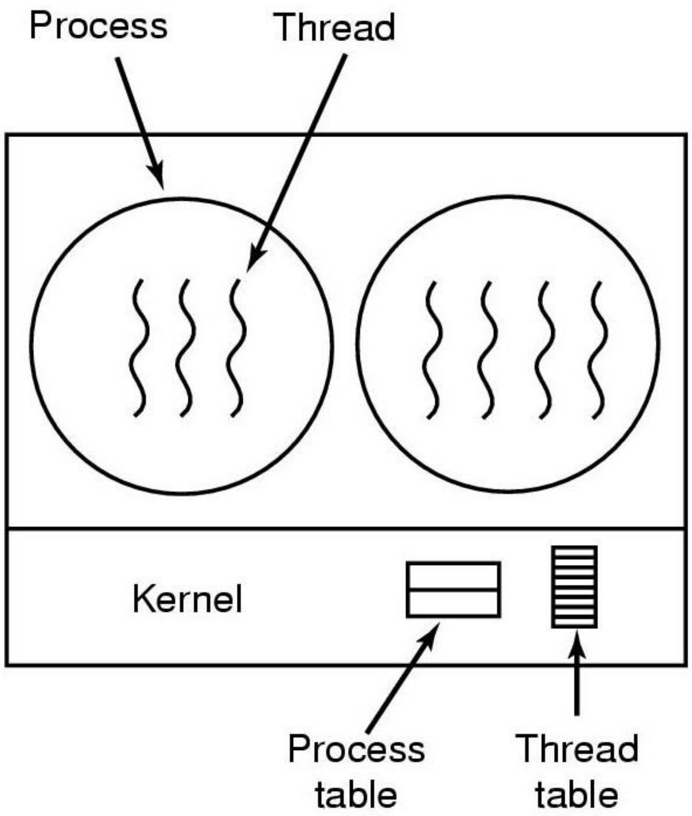
User-Level Threads

The thread-switching code is in user space



Kernel-Level Threads

Thread-switching code is in the kernel



Thanks