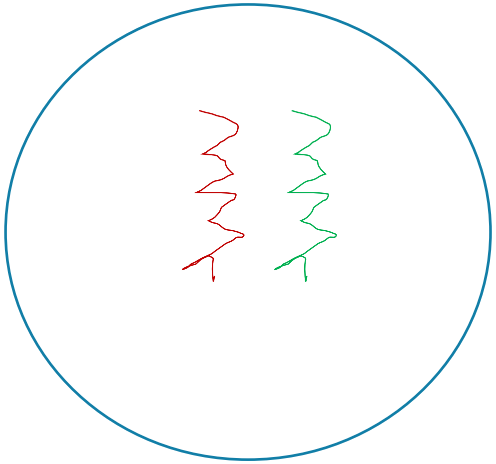
CSci 4061
Introduction to Operating Systems

Module 5: Threads

(Thread-Basics)

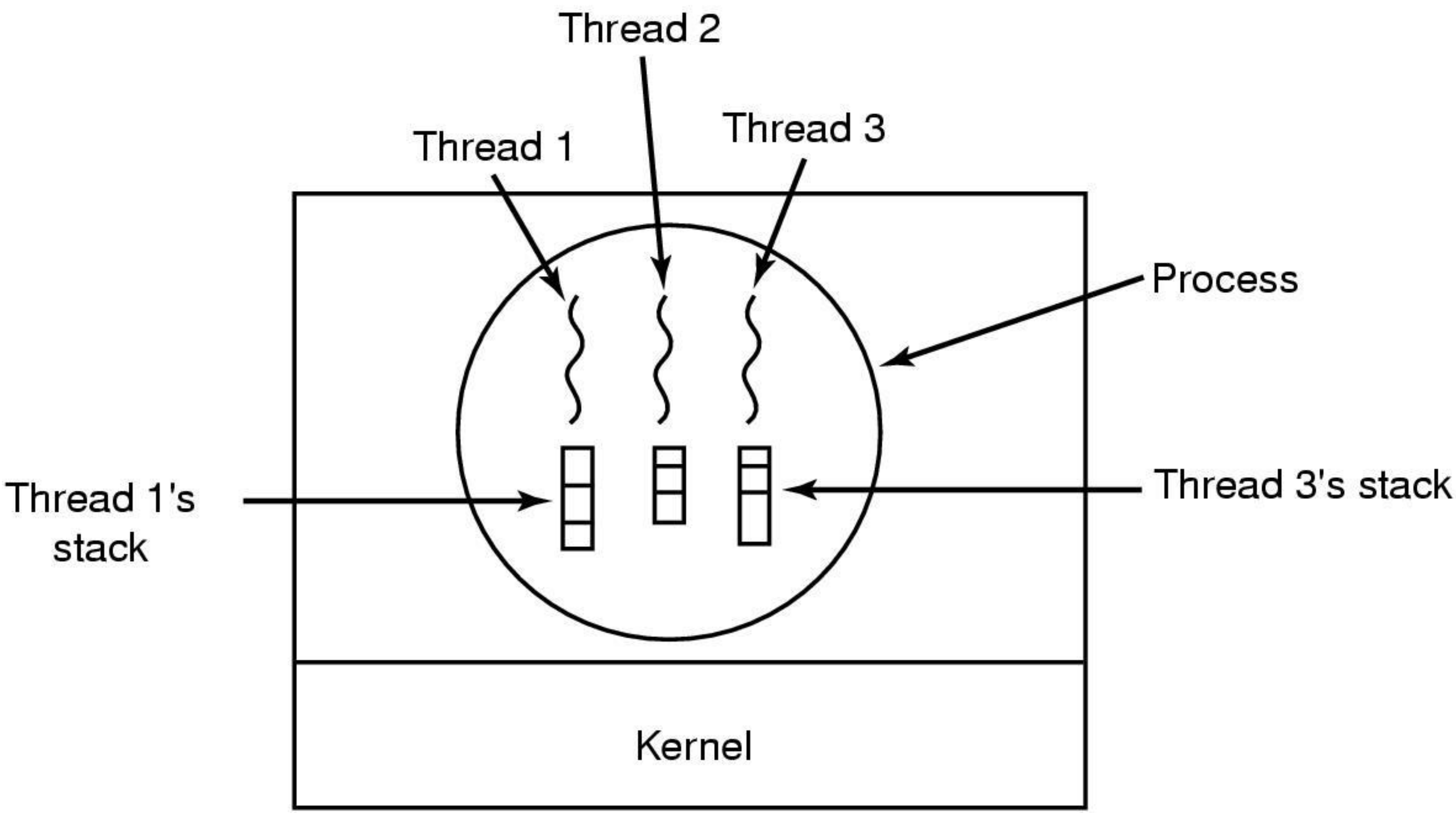
Chapter 12

Threads

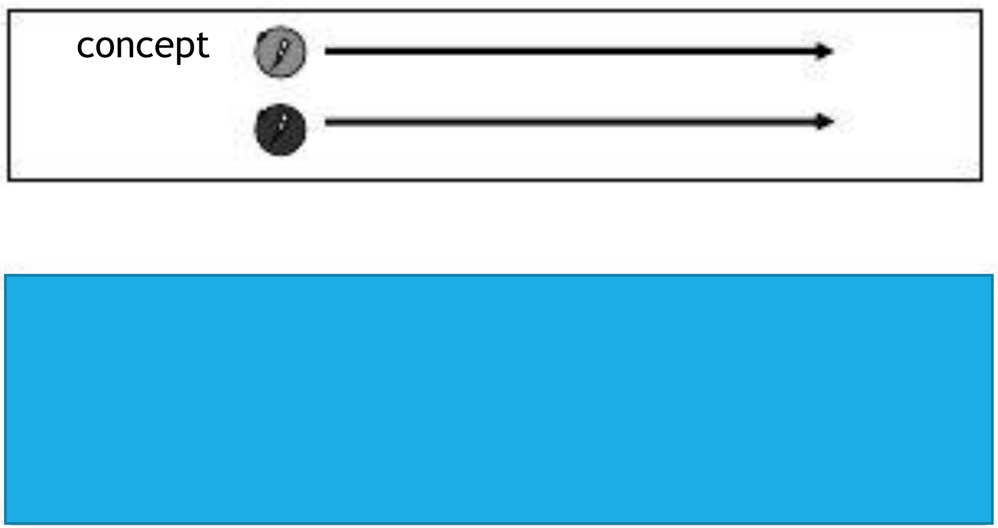


- Abstraction: for an executing instruction stream
- Threads exist within a process and share its resources (i.e. memory)
- But, thread has its own stack and "PC"
- Default: always 1 thread (implicit)

Another View



Two Threads Sharing a CPU

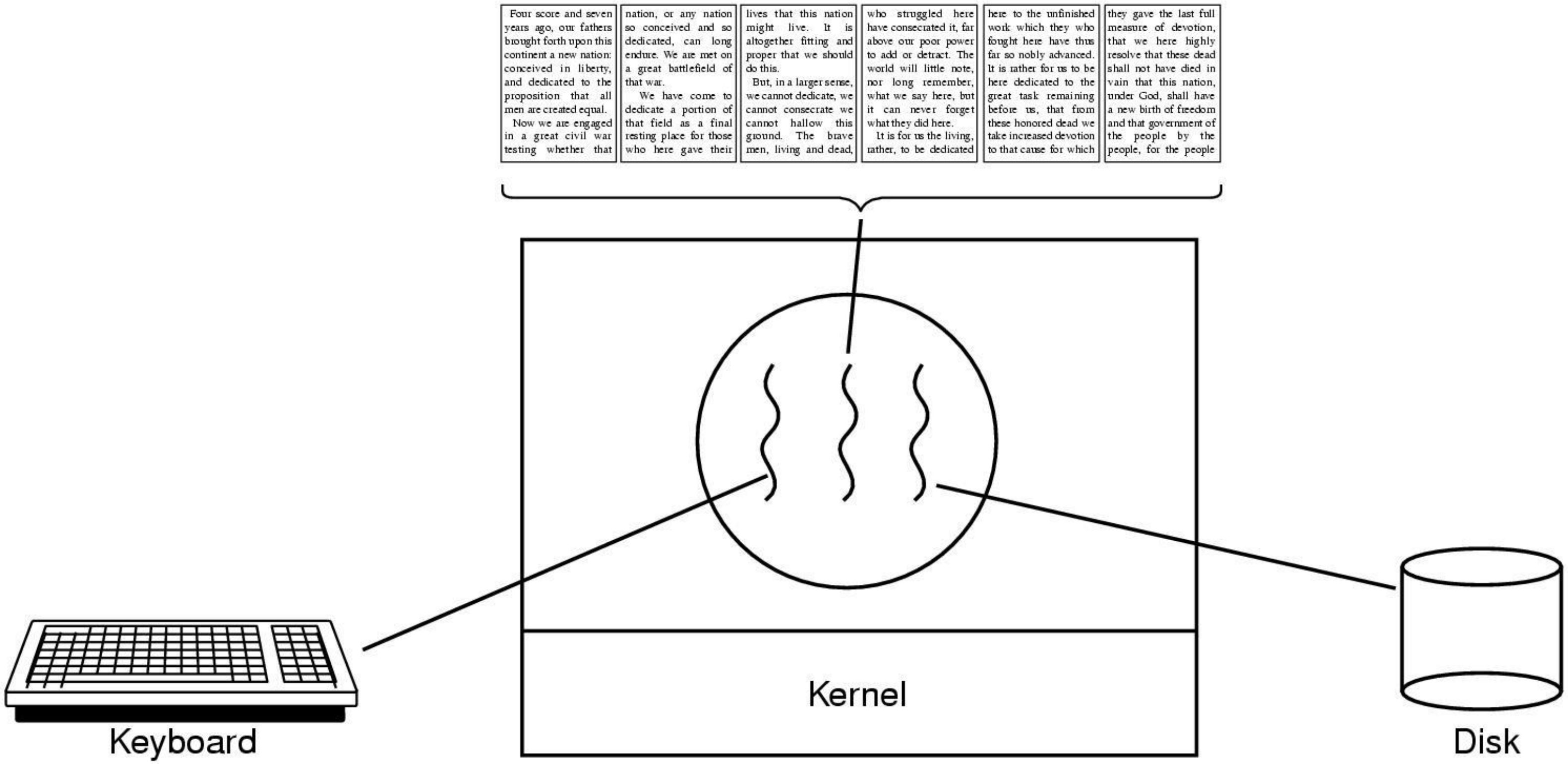


What may cause a switch?

Thread Benefits

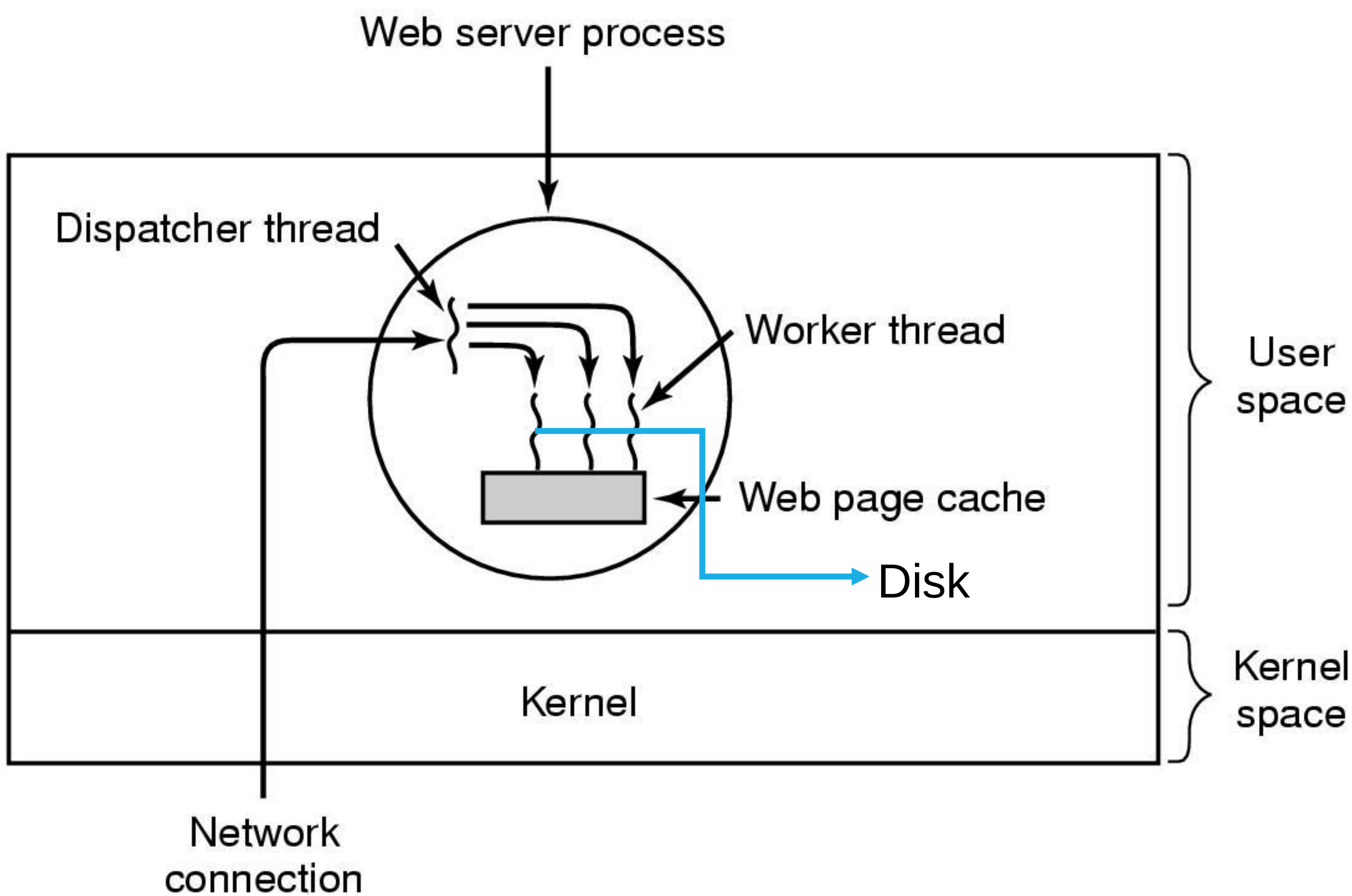
- Concurrency
  - when one blocks, another runs
- Modularity
  - decompose functionality
- Parallelism
  - threads running in parallel – multi-core
- Scale
  - more threads than processors/processes
- Overhead
  - cheaper than processes

Threads Example: editor



When one blocks, another can run …

Thread example: Web server…



When one blocks, another can run …

How to Quantify the Benefit?

- Web request: read request from network, fetch page, write request to network
- $T_{req} = T_{read} + T_{serve} + T_{write}$
- $T_{req} = T_{read} + h*T_{cache} + (1-h)*T_{disk} + T_{write}$

- Can we improve the performance of a single request with threads?
- Can we improve the performance of the "service", requests/time?

Thread Example: Web server

dispatcher (...) {

```
while (TRUE){
 // 1. read (req ~ URL)
 get_next_request (&req);
 handoff_work (&req, &buf);
}
```

worker (...) {

```
wait_for_work (&buf, &req)
// 2. service
look_for_page_in_cache (&req, &answer);
if (page_not_in_cache (&answer) {
  read_page_from_disk (&req, &answer);
  put_page_in_cache (&req, &answer);
}
// 3. write
return (&answer);
```

- How are these threads interacting?
  - Shared memory: threads share buffer, cache
  - Threads share globals, heap, NOT stack

Looks great

- Drawbacks?
- Alternatives?

Drawbacks

- Sharing
  - Synchronization is needed to protect shared data structures: Web server? Editor?
  - Assume: threads may be switched unpredictably!
  - Failure: no isolation

- Thread-safety (related to Sharing)
  - Not all system calls may be **thread-safe**
  - System call (or any call) that can be executed concurrently by multiple threads

- Global variables
  - Per thread globals may be needed

Drawbacks: Sharing/Thread-safe

```
int counter = 0;
int increment_counter (){
    counter ++;  // counter = counter +1
return counter;
}
```

problem? Suppose threads T1 and T2 call it

Unrolling counter = counter +1

load counter->R

incr R

store R->counter

Thread Safety (cont’d)

```
int counter = 0
lock_type counter_lock;
int increment_counter (){
      // lock is held or free: if held, caller is blocked
      lock (counter_lock);
      counter++;
      unlock (counter_lock);
      return counter;
}
```

Locks

- Just to be safe, shouldn’t I always put locks around my code?
  - Locks reduce concurrency and performance, use only when needed!

Drawbacks:per thread globals

T1

…

```
syscall_1
sets errno
```

… {T1 switches to T2}

```
reads errno
```

T2

…

```
syscall_2
sets errno
```

…

- In Unix, *errno* is a global variable in shared library
- Options to guarantee error reporting is thread-safe?
    - Use locks
    - Eliminate global: return error code
    - Define errno “service” or macro

```
#define errno _special_thread_errno (thread_id)
```

Alternatives to Threads

- Want concurrency
  - Goal: If a program (or part of a program) cannot make progress due to blocking, then allow:

    some other part of the program to make progress
- Options?