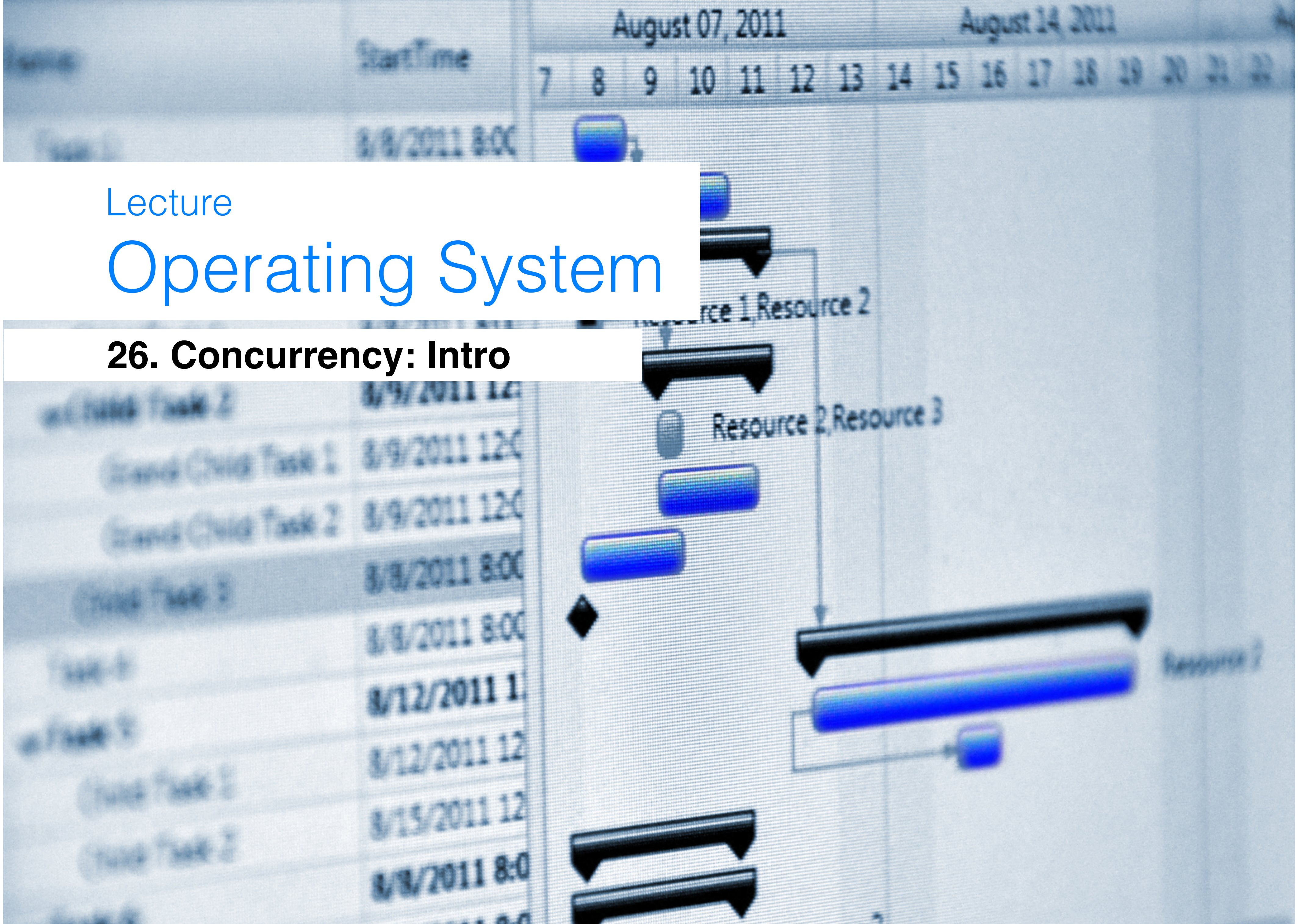
Lecture

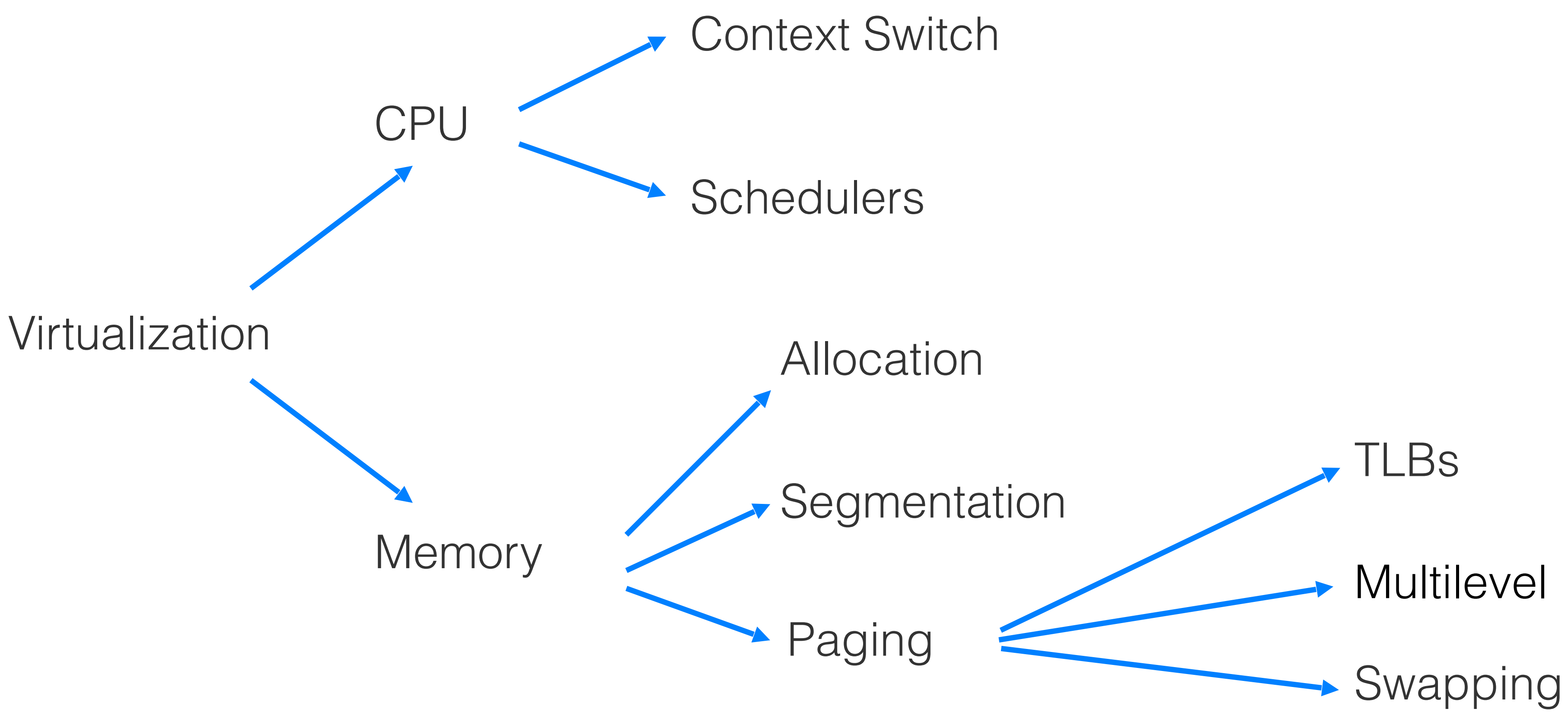
Operating System

26. Concurrency: Intro



Professor Dr. Michael Mächtel

Review: Easy Piece 1



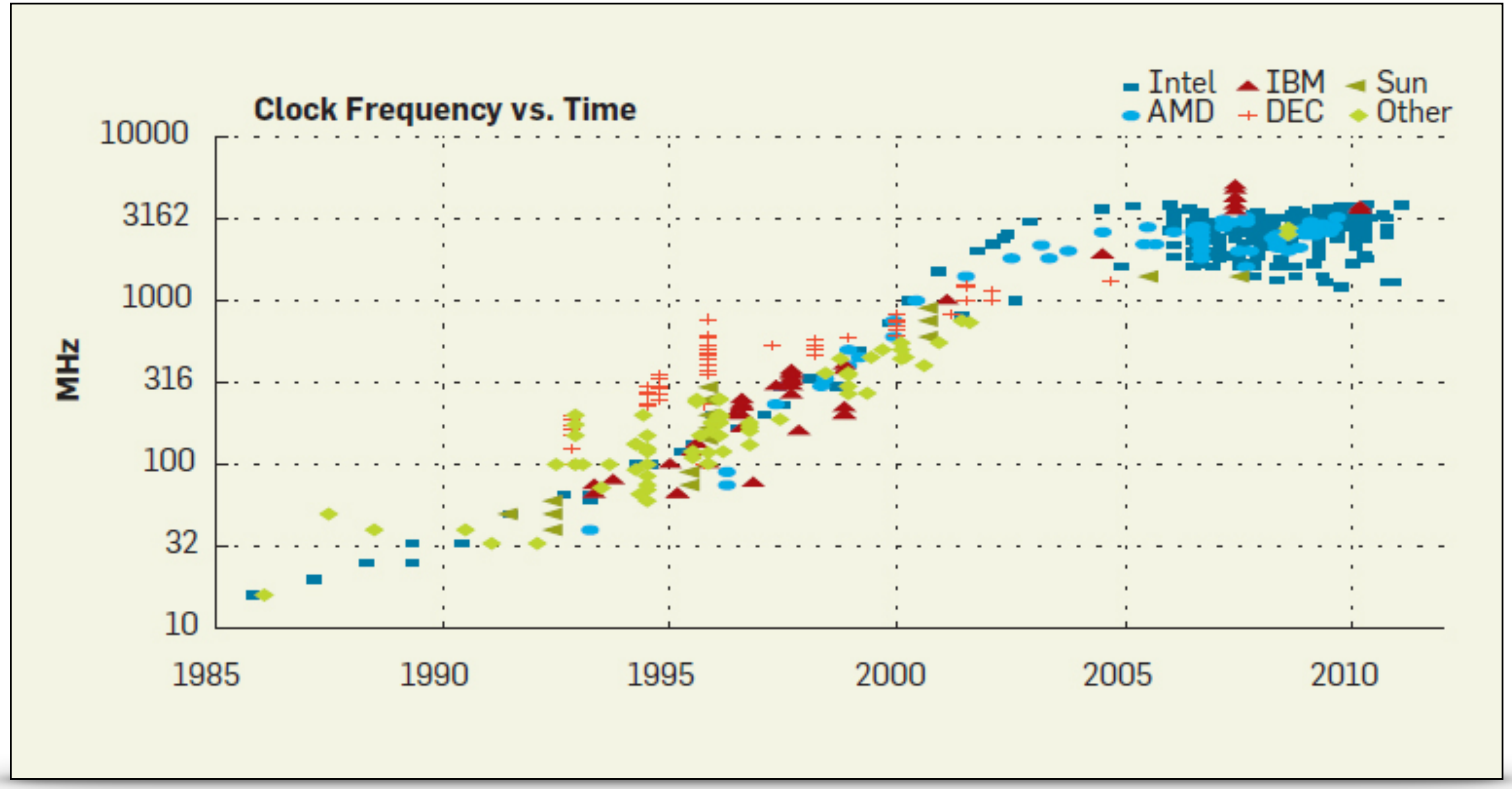
26. Concurrency: An Introduction

1. **Motivation**
2. **Thread**
3. **OS Support for Threads**
4. **Concurrency Terms**

26. Concurrency: An Introduction

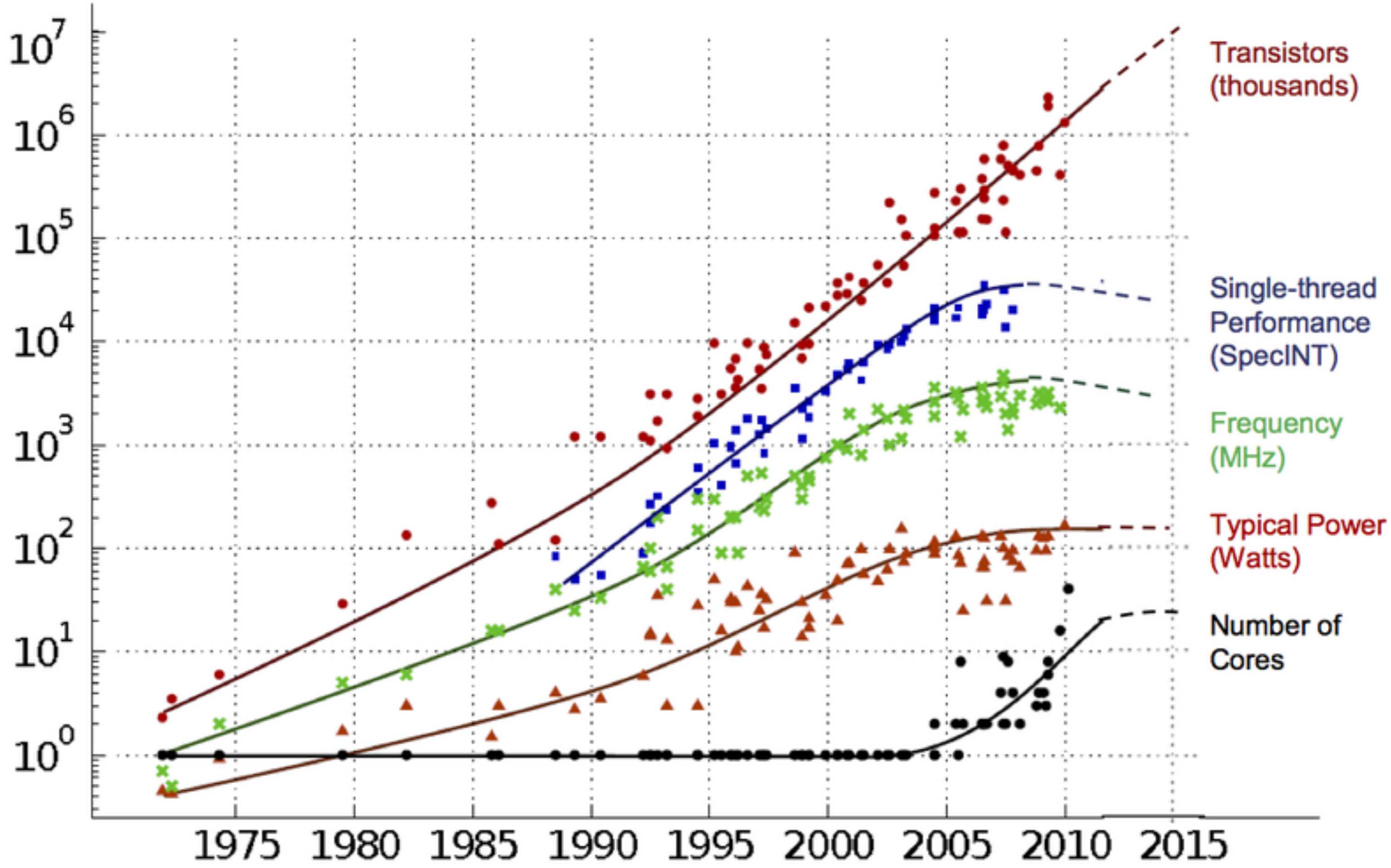
1. **Motivation**
2. Thread
3. OS Support for Threads
4. Concurrency Terms

Motivation for Concurrency



http://cacm.acm.org/magazines/2012/4/147359-cpu-db-recording-microprocessor-history/fulltext

Motivation for Concurrency (Cont.)



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Motivation

- CPU Trend: Same speed, but multiple cores
- Goal: Write applications that fully utilize many cores
- **Option 1**: Build apps from many communicating processes
  - Example: Chrome (process per tab)
  - Communicate via pipe() or similar
- Pros?
  - Don’t need new abstractions; good for security
- Cons?
  - Cumbersome programming
  - High communication overheads
  - Expensive context switching (why expensive?)

Motivation

- **Option 2:** New abstraction: thread
- Threads are like processes, except:
    - multiple threads of same process share an address space
- Divide large task across several cooperative threads
- Communicate through shared address space

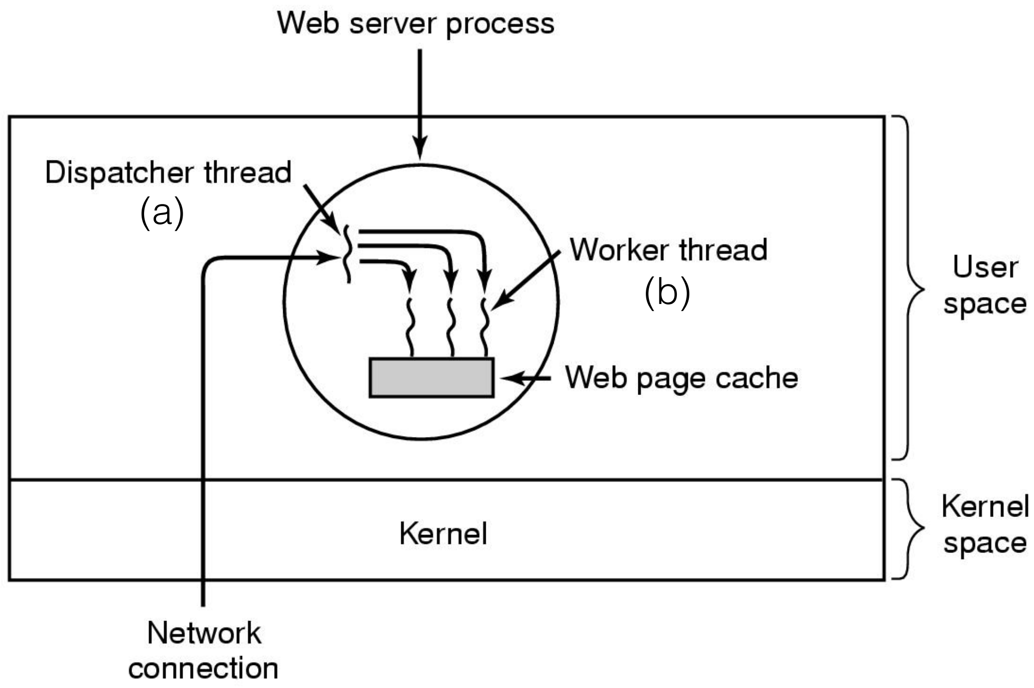
26. Concurrency: An Introduction

1. Motivation
2. **Thread**
3. OS Support for Threads
4. Concurrency Terms

Thread

- A new abstraction for a single running process
- Multi-threaded program:
  - A multi-threaded program has more than one point of execution.
  - Multiple PCs (Program Counter)
  - They **share** the same **address space**.

Example: Webserver



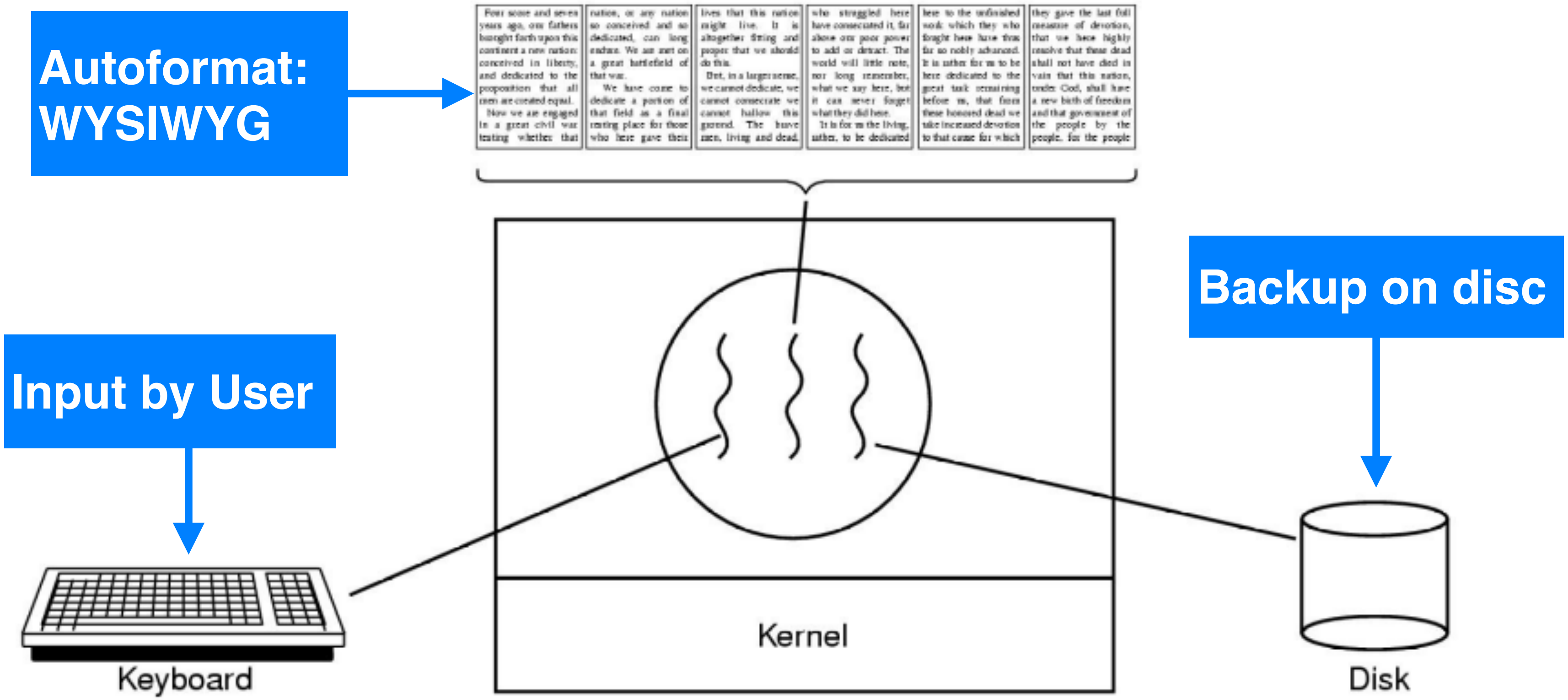

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(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)

Example: Wordprocessing

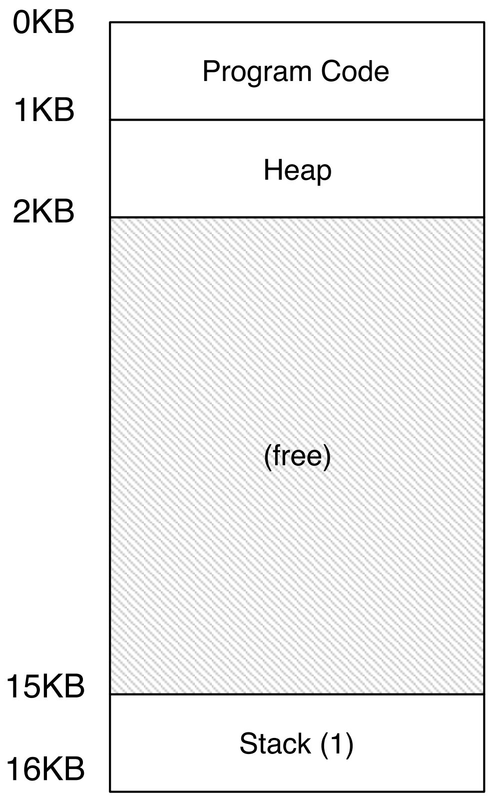


Context switch between threads

- Each thread has its own program counter and set of registers.
  - One or more **thread control blocks(TCBs)** are needed to store the state of each thread.
- When switching from running one (T1) to running the other (T2),
  - The register state of T1 be saved.
  - The register state of T2 restored.
  - The **address space remains** the same.

The stack of the relevant thread

- There will be **one stack per thread**.

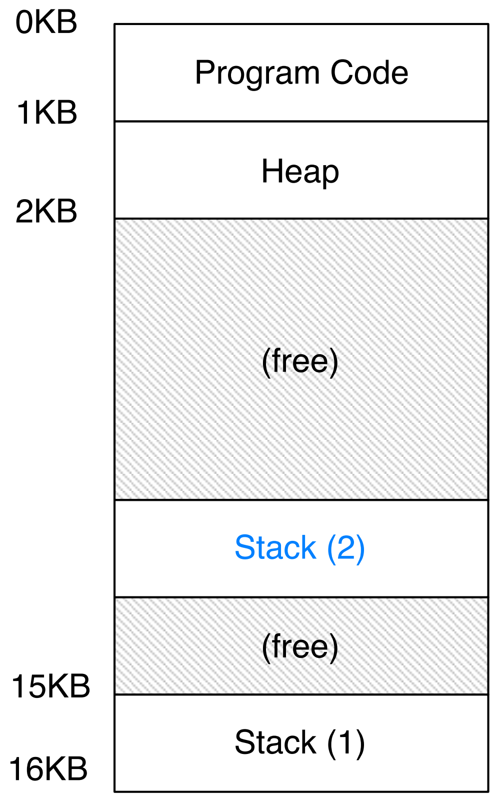


The code segment:
where instructions live

The heap segment:
contains malloc'd data
dynamic data structures
(it grows downward)

(it grows upward)
The stack segment:
contains local variables
arguments to routines,
return values, etc.

A Single-Threaded Address Space



Two threaded Address Space

Thread vs. Process

- Multiple threads within a single process share:
  - Process ID (PID)
  - Address space
    - Code (instructions)
    - Most data (heap)
  - Open file descriptors
  - Current working directory
  - User and group id
- Each thread has its own
  - Thread ID (TID)
  - Set of registers, including program counter and stack pointer
  - Stack for local variables and return addresses (in same address space)

Thread API

- Variety of thread systems exist
  - POSIX Pthreads
- Common thread operations
  - Create
    - `pthread_create()`
  - Exit
    - `pthread_exit()`
  - Join (instead of `wait()` for processes)
    - `pthread_join()`

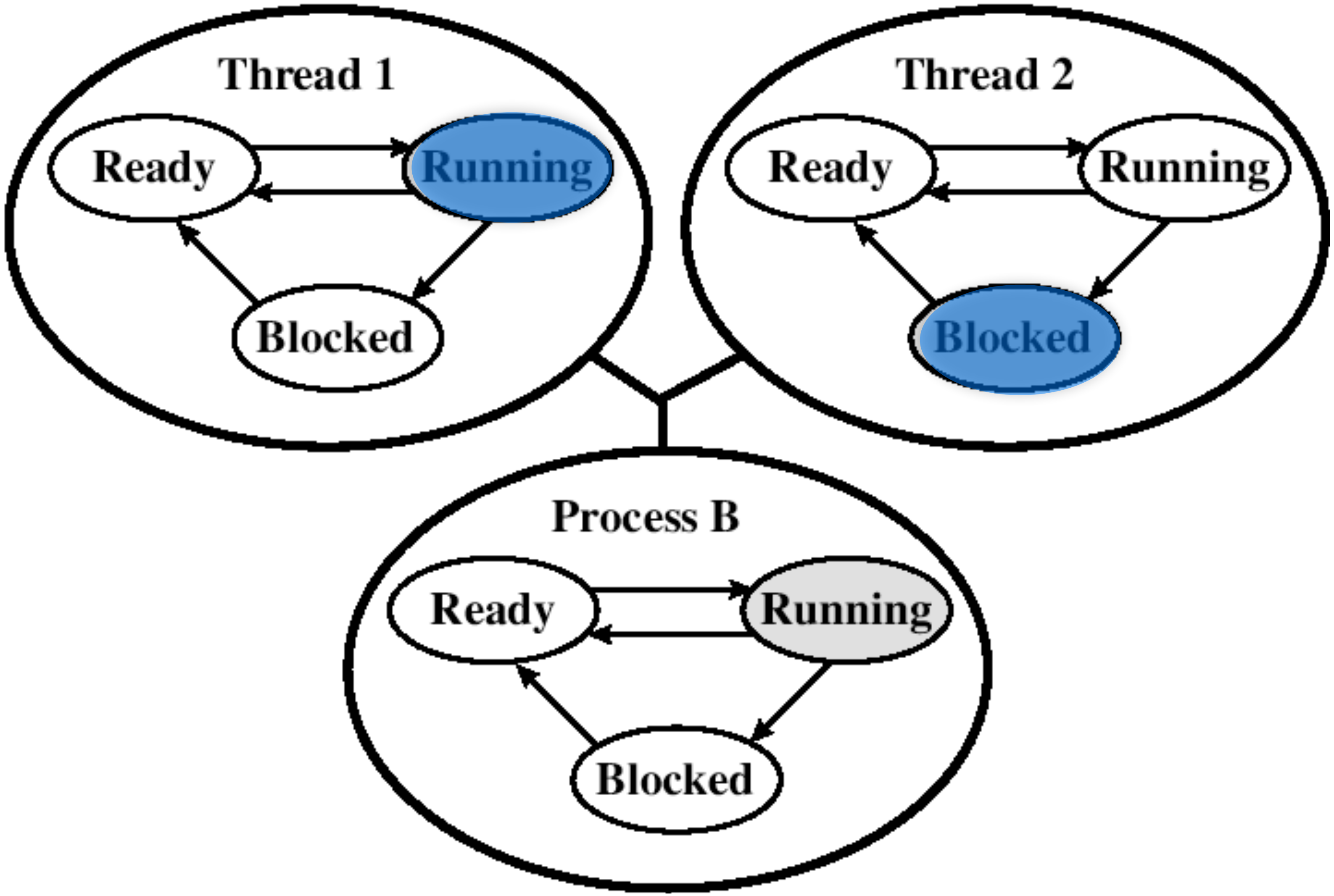
26. Concurrency: An Introduction

1. Motivation
2. Thread
3. **OS Support for Threads**
4. Concurrency Terms

OS Support: Approach 1

- **User-level threads**: Many-to-one thread mapping
  - Implemented by user-level runtime libraries
    - Create, schedule, synchronize threads at user-level
  - OS is not aware of user-level threads
    - OS thinks each process contains only a single thread of control
- Advantages
  - Does not require OS support; Portable
  - Can tune scheduling policy to meet application demands
  - Lower overhead thread operations since no system call
- Disadvantages
  - Cannot leverage multiprocessors
  - Entire process blocks when one thread blocks

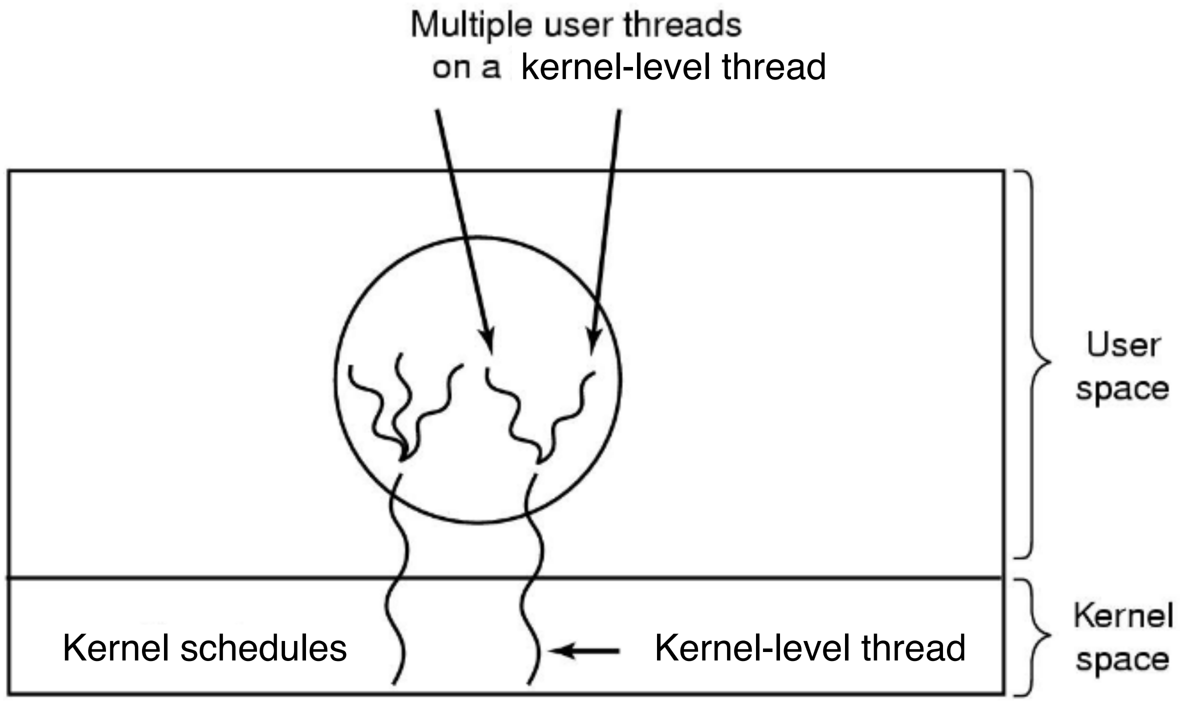
User-Level Thread States



OS Support: Approach 2

- **Kernel-level threads:** One-to-one thread mapping
  - OS provides each user-level thread with a kernel-level thread
  - Each kernel-level thread scheduled independently
  - Thread operations (creation, scheduling, synchronization) performed by OS
- Advantages
  - Each kernel-level thread can run in parallel on a multiprocessor
  - When one thread blocks, other threads from process can be scheduled
- Disadvantages
  - Higher overhead for thread operations
  - OS must scale well with increasing number of threads

OS Support: Hybrid Approach



Important: Kernel Level Threads != Kernel Threads

26. Concurrency: An Introduction

1. Motivation
2. Thread
3. OS Support for Threads
4. **Concurrency Terms**

Concurrency Example

Using POSIX Wrappers

- The main program creates **two** threads.
- **Thread**: Each thread start running in a routine called `worker()`.
- **`worker()`**: increments a counter
- **`loops`** determines how many times each of the two workers will increment the shared counter in a loop.

```
volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads
                              <loops>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

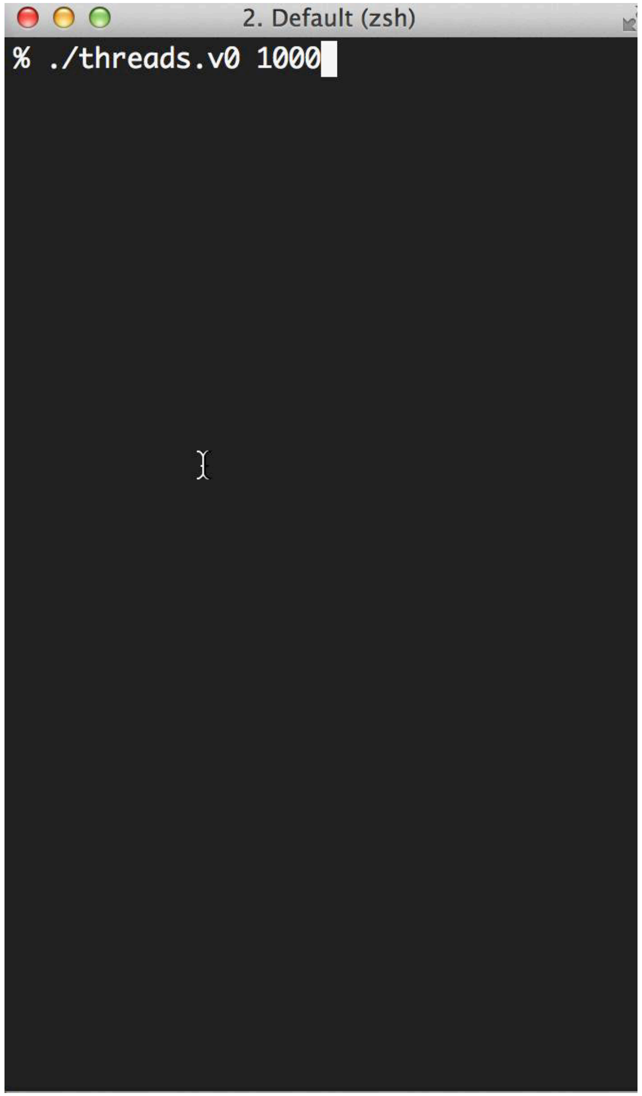
Concurrency Example

Using POSIX Wrappers

```c
volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads
                         <loops>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```



Race condition

- Example with two threads
  - counter = counter + 1 (default is 50)
  - We expect the result is 52. However,

OS	Thread1	Thread2	PC (after instruction)	%eax (after instruction)	counter (after instruction)
	`before counter += 1`		100	0	50
	`mov 0x8049a1c, %eax`		105	50	50
	`add $0x1, %eax`		108	51	50
`interrupt`					
`save T1's state`					
`restore T2's state`			100	0	50
		`mov 0x8049a1c, %eax`	105	50	50
		`add $0x1, %eax`	108	51	50
		`mov %eax, 0x8049a1c`	113	51	51
`interrupt`					
`save T2's state`					
`restore T1's state`			108	51	50
	`mov %eax, 0x8049a1c`		113	51	**51**

Critical section

- A piece of code that **accesses a shared variable** and must not be concurrently executed by more than one thread.
  - Multiple threads executing critical section can result in a race condition.
  - Need to support **atomicity** for critical sections **(mutual exclusion**)

Non-Determinism

- **Concurrency** leads to non-deterministic results
  - Not deterministic result: different results even with same inputs
  - race conditions
- Whether bug manifests depends on CPU schedule!
- Passing tests means little!
- How to program:
  - imagine scheduler is malicious
  - Assume scheduler will pick bad ordering at some point…

What do we want?

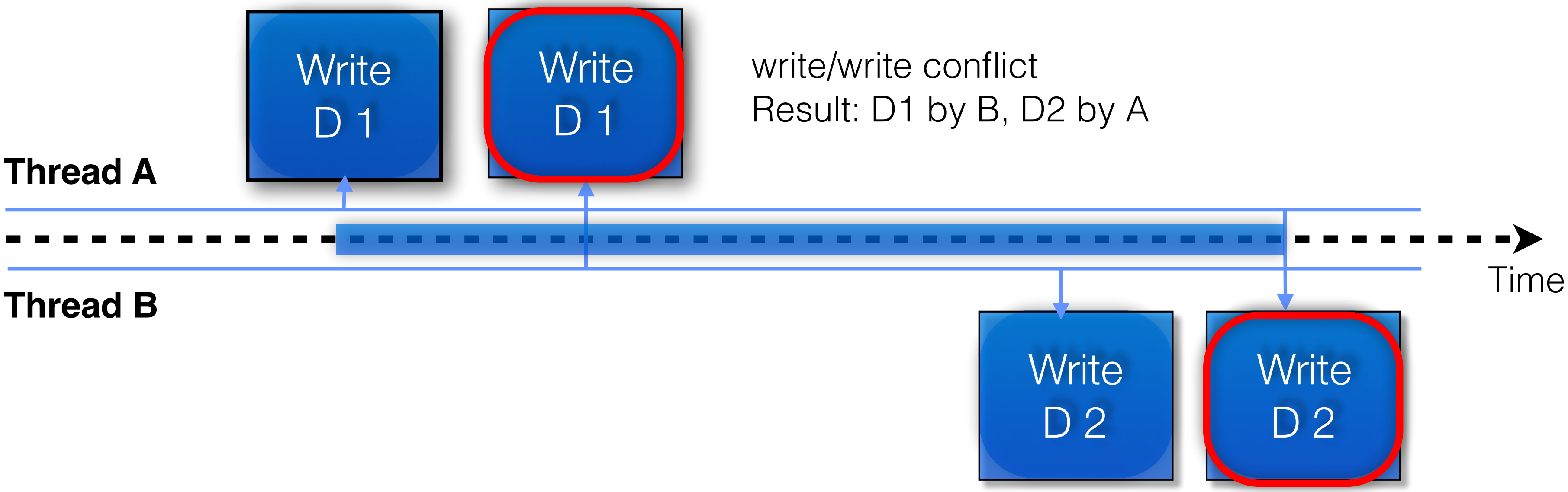
- Want 3 instructions to execute as an uninterruptable group
- That is, we want them to be **atomic**

```
mov 0×123, %eax
add %0×1, %eax
mov %eax, 0×123
```

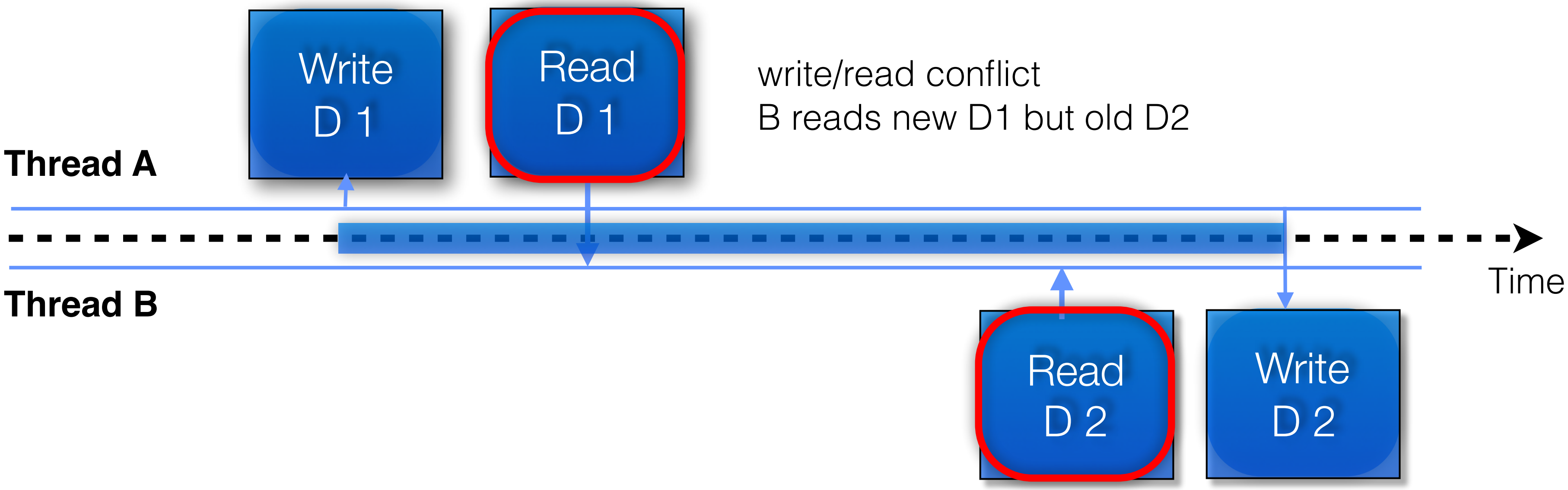
critical section

- More general:
  - Need mutual exclusion for critical sections
    - if process A is in critical section C, process B can’t
    - (it's okay, if other processes do unrelated work)

Write/Write and Write/Read Conflict



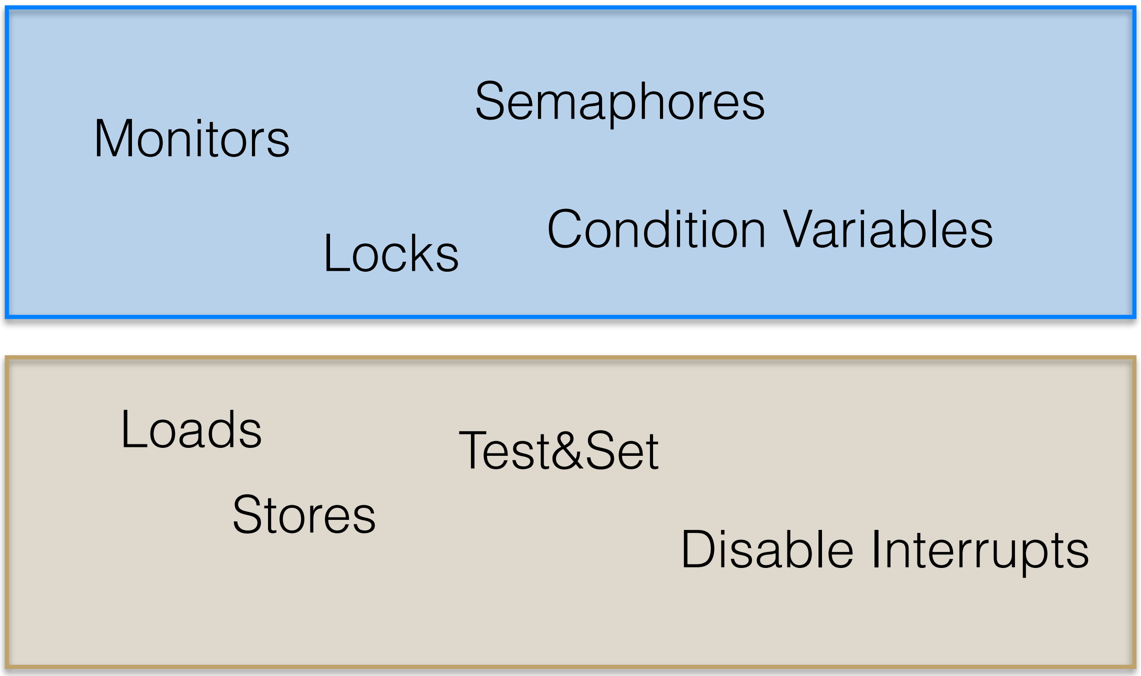
write/write conflict
Result: D1 by B, D2 by A



write/read conflict
B reads new D1 but old D2

Synchronization

- Build higher-level synchronization primitives in OS
  - Operations that ensure correct ordering of instructions across threads
- Motivation: Build them once and get them right



Locks

- Ensure that any such critical section executes as if it were a single atomic instruction (**execute a series of instructions atomically**).
- Goal: Provide **mut**ual **ex**clusion (mutex)
- Three common operations for locks:
  - Allocate and Initialize
  - Acquire
    - Acquire exclusion access to lock;
    - Wait if lock is not available (some other process in critical section)
    - Spin or block (relinquish CPU) while waiting
  - Release
    - Release exclusive access to lock; let another process enter critical section

Key Concurrency Terms

- **Critical section**
  - Piece of code, that accesses a shared resource.
- **Race condition**
  - arises, if multiple threads enter the critical section, leading to non-deterministic results.
- **Indeterminate programm** consits of one or more race conditions. The outcome is non-deterministic.
- **Mutual Exclusion**
  - garantees, that only a single thread enters a critical section, thus avoiding races, resulting in deterministic outputs.

Why in OS Class?

- History
  - OS was the first concurrent program
  - many techniques were created for use within
  - later multi-threaded apps had to consider same
- Because an interrupt may occur at any time
  - code that updates critical data = critical section
  - internal critical data:
    - page tables
    - process lists
    - file system structures
    - virtual every kernel data structure .....

Thanks

Questions?