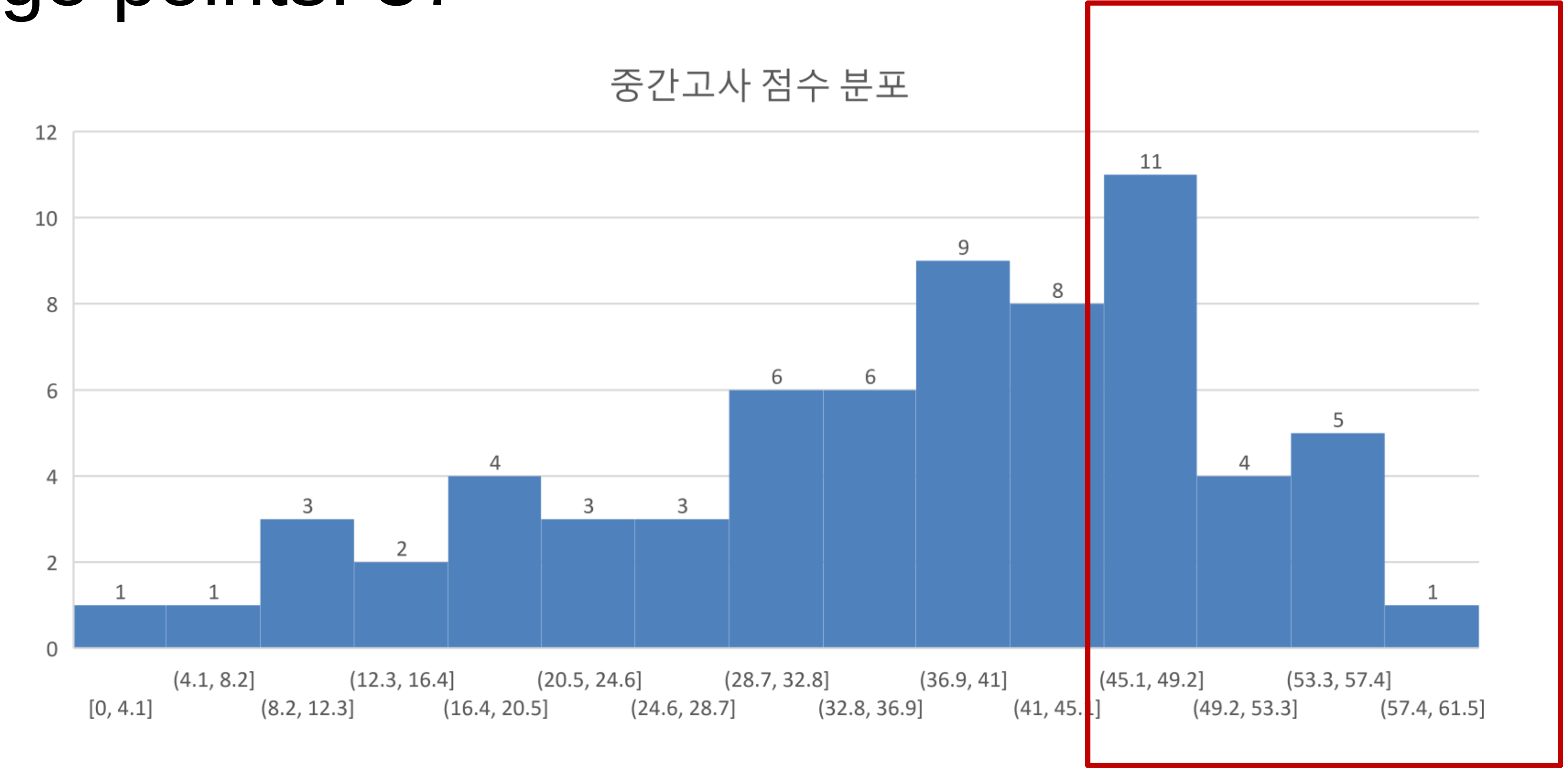
Operating Systems

Thread

Young-Woo Kwon

Midterm result

- Total: 80 + Bonus 10
- Highest points: 61.5
- Average points: 37



MEMORY
(PAGING AND VM)

malloc()/free() vs. GC

Explicit Alloc/Dealloc

- Advantages:
  - Typically faster than GC
  - No GC “pauses” in execution
  - More efficient use of memory
- Disadvantages:
  - More complex for programmers
  - Tricky memory bugs
    - Dangling pointers
    - Double-free
    - Memory leaks
  - Bugs may lead to security vulnerabilities

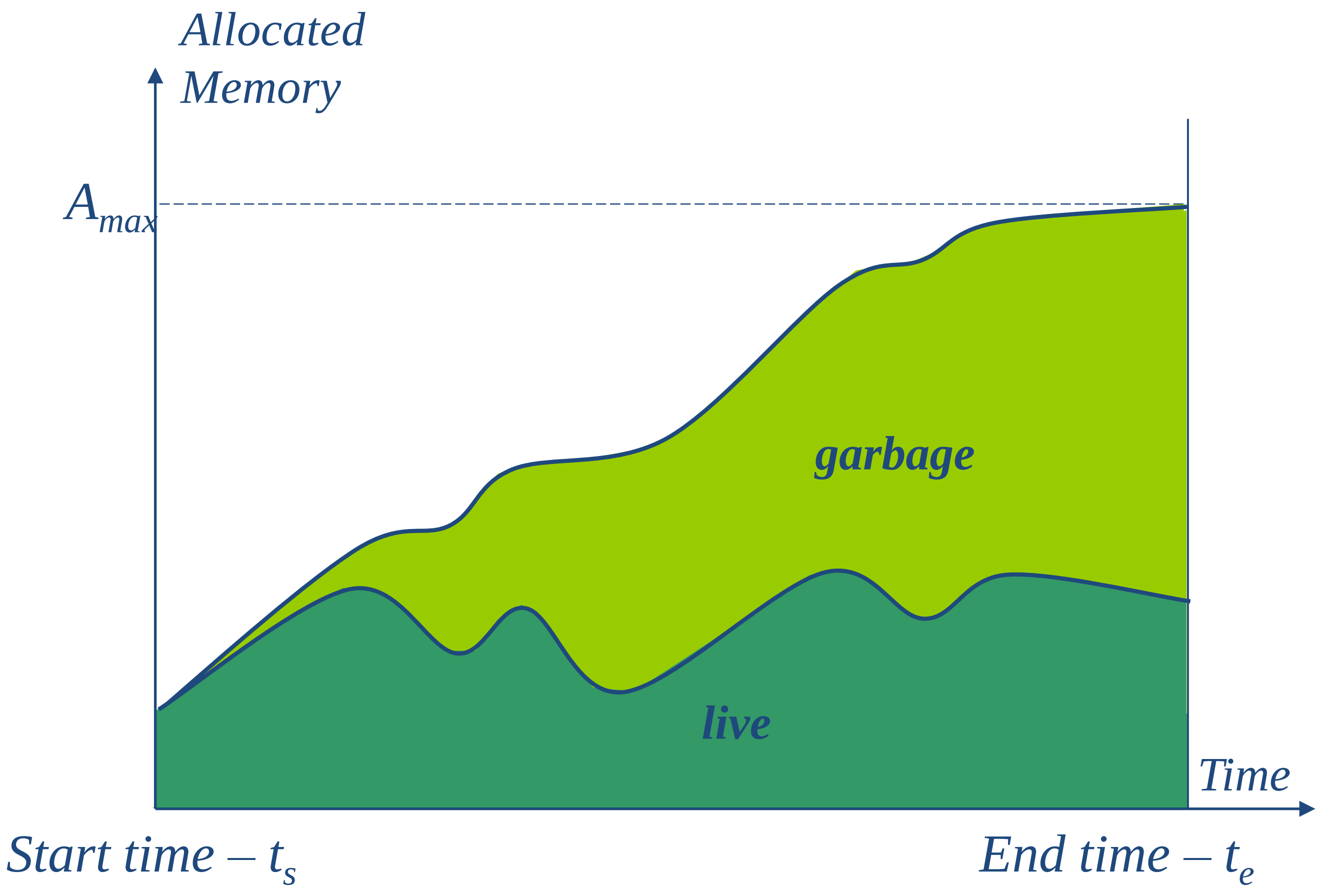
Garbage Collection

- Advantages:
  - Much easier for programmers
- Disadvantages
  - Typically slower than explicit alloc/dealloc
  - Good performance requires careful tuning of the GC
  - Less efficient use of memory
  - Complex runtimes may have security vulnerabilities
    - JVM gets exploited all the time

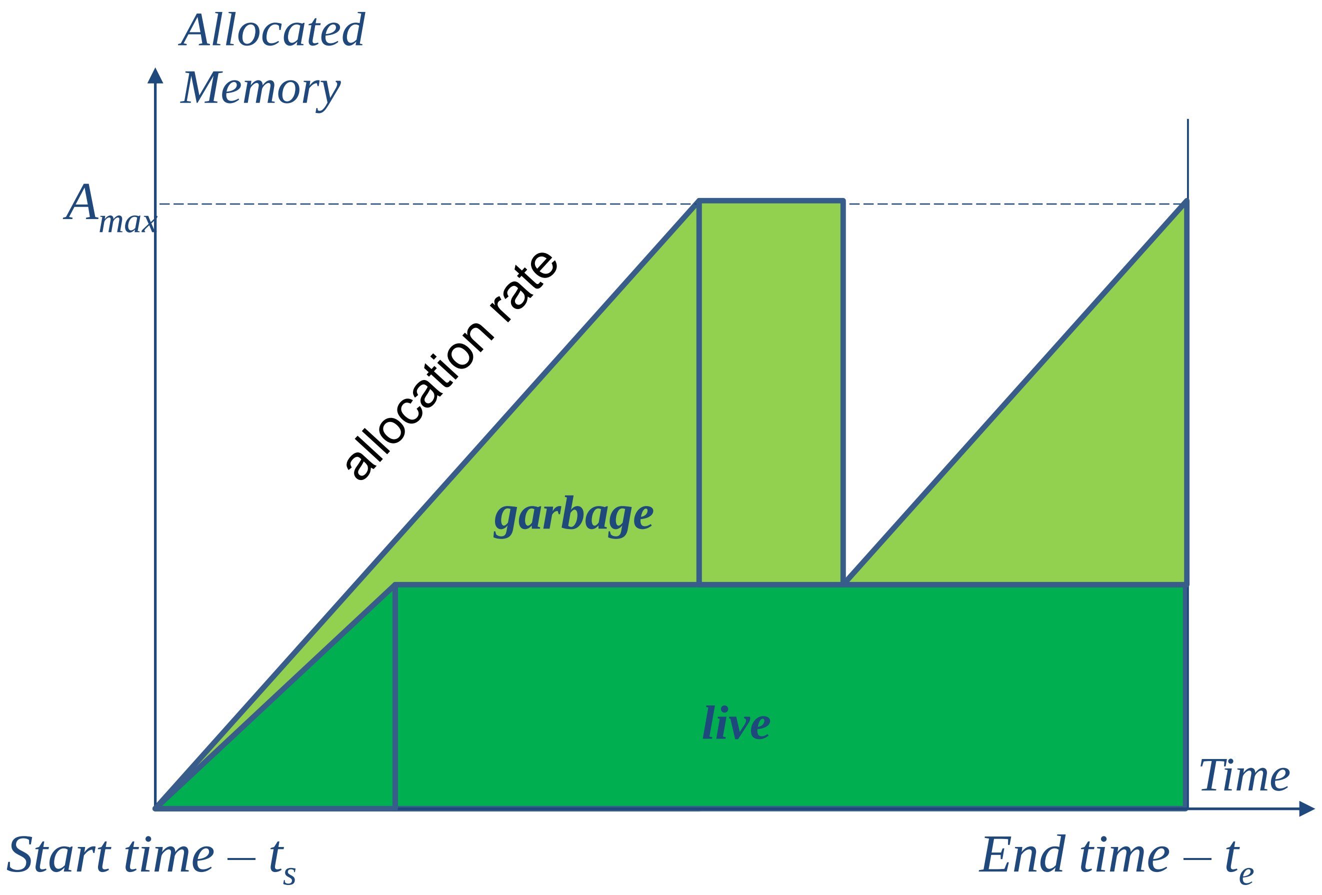
Heap Size vs. GC Frequency

- All else being equal, smaller maximum heap sizes necessitate more frequent collections
  - Old rule of thumb: need between 1.5x and 2.5x times the size of the live heap to limit collection overhead to 5-15% for applications with reasonable allocation rates
  - [Hertz 2005] finds that GC outperforms explicit MM when given 5x memory, is 17% slower with 3x, and 70% slower with 2x
  - Performance degradation occurs when live heap size approaches maximum heap size

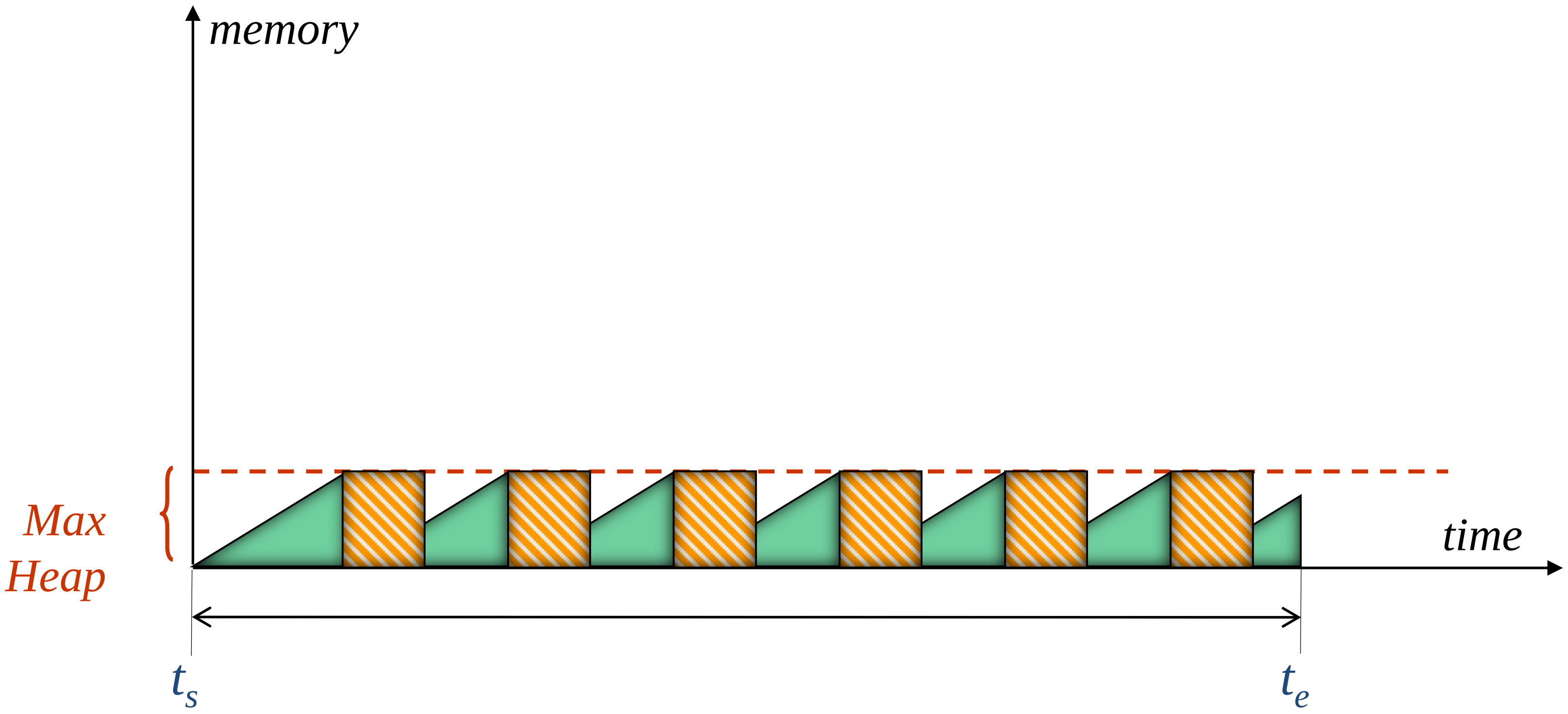
Memory Allocation Time-Profile



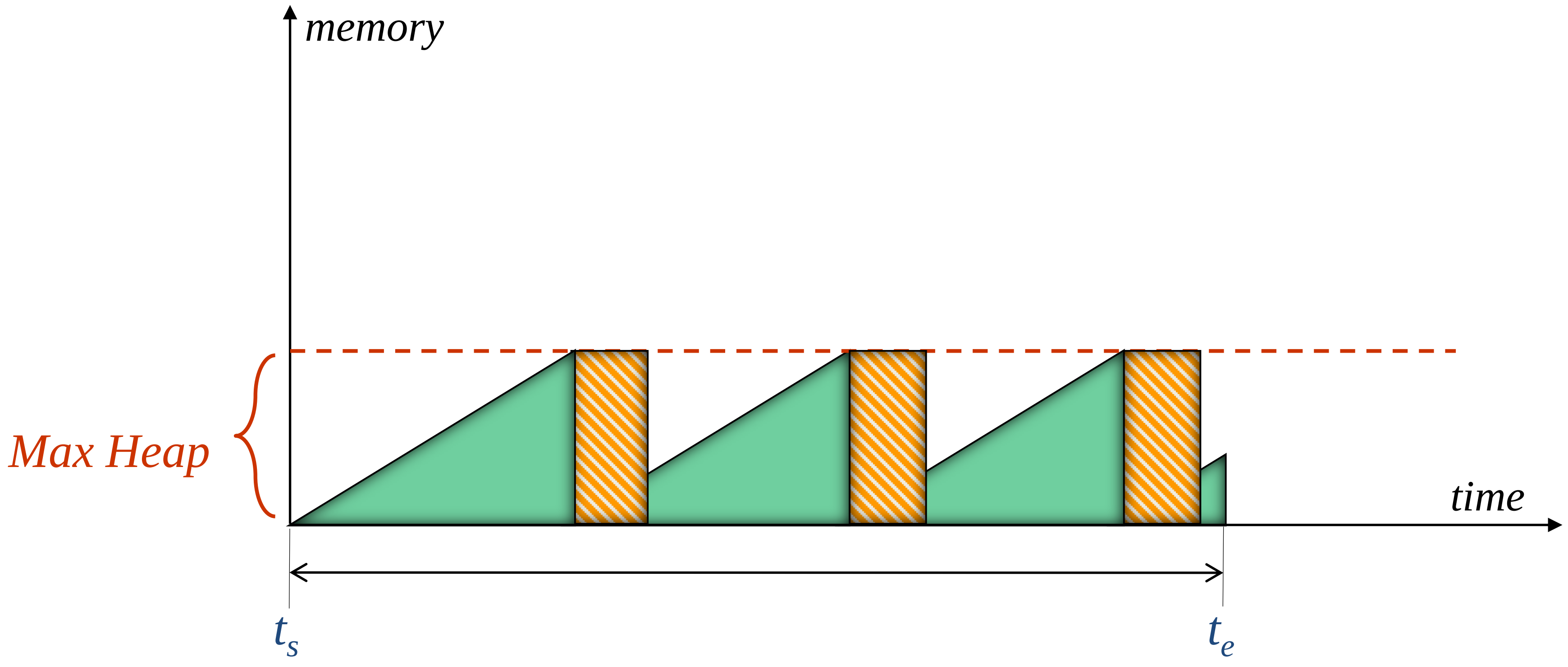
Modeling Memory Allocation



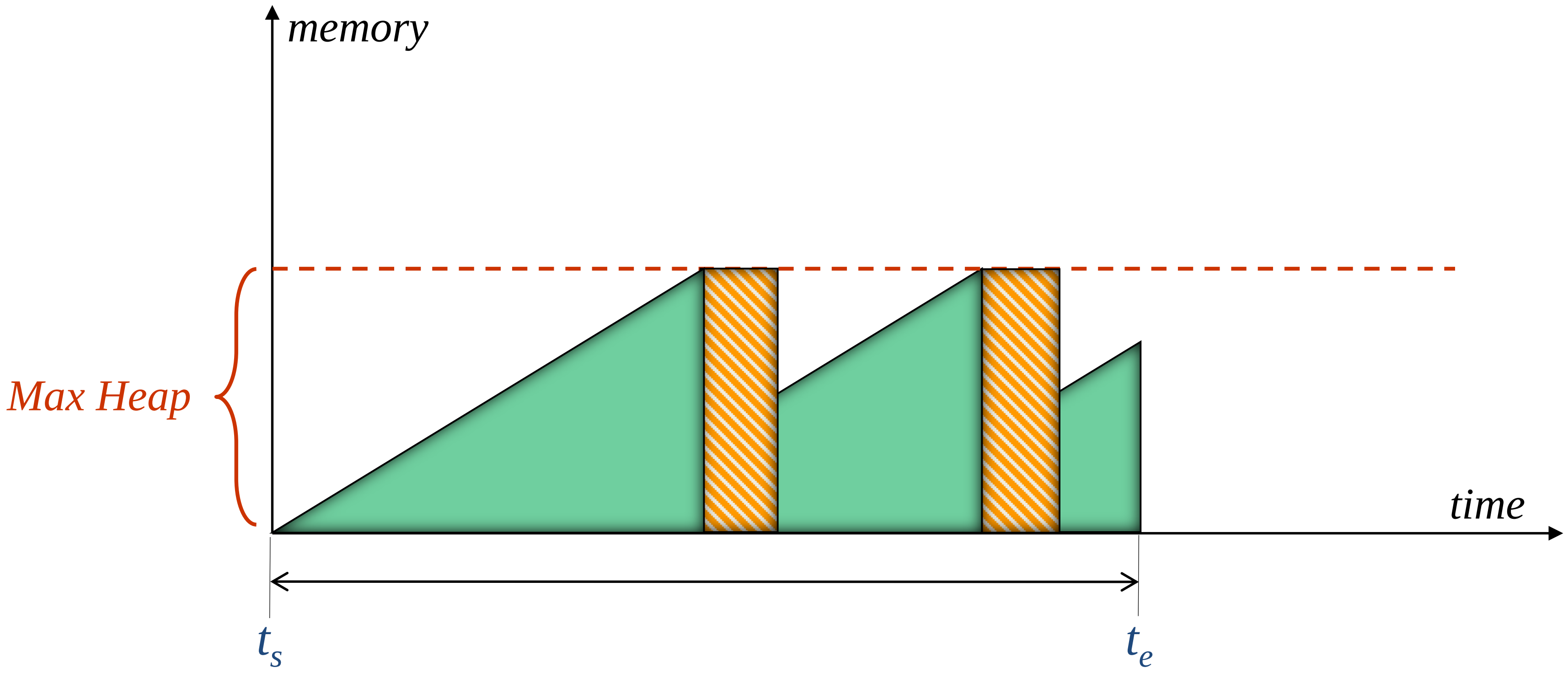
Execution Time vs. Memory



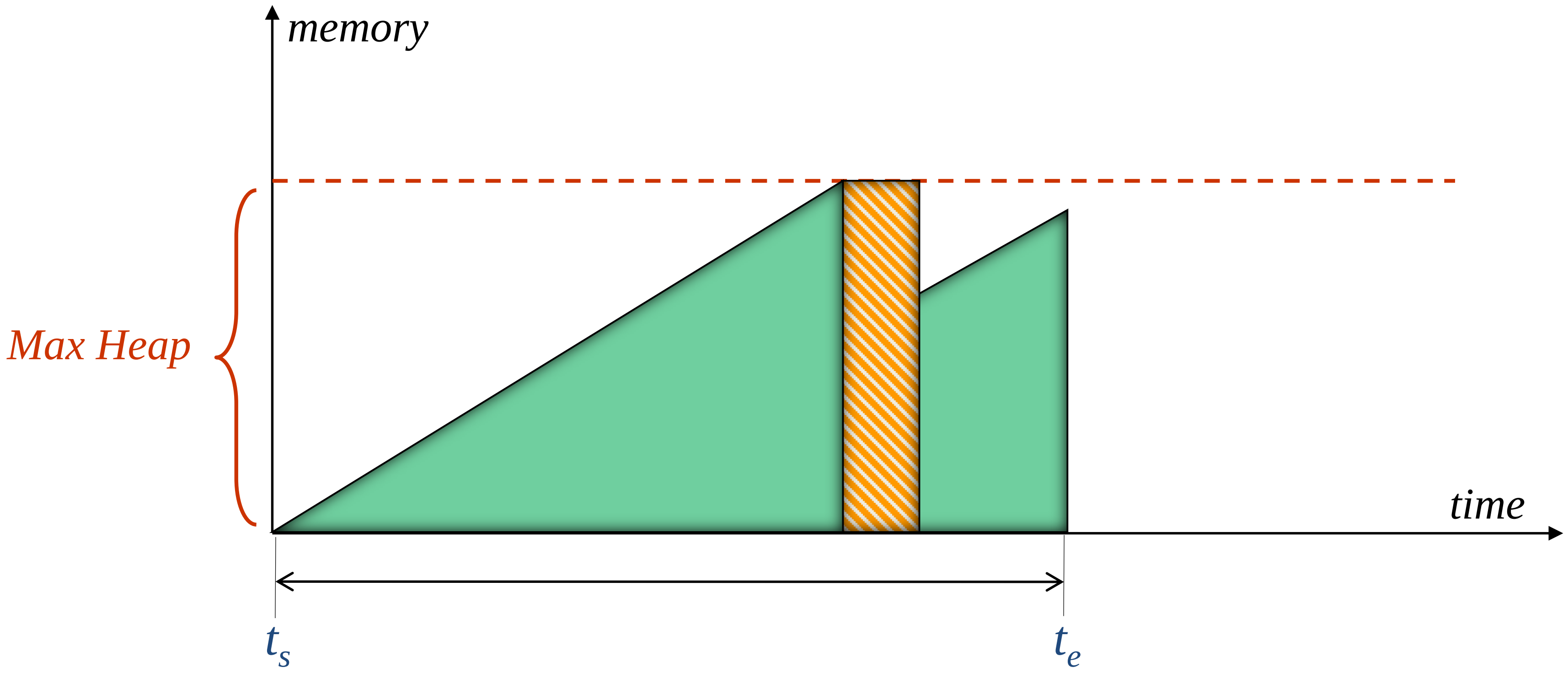
Execution Time vs. Memory



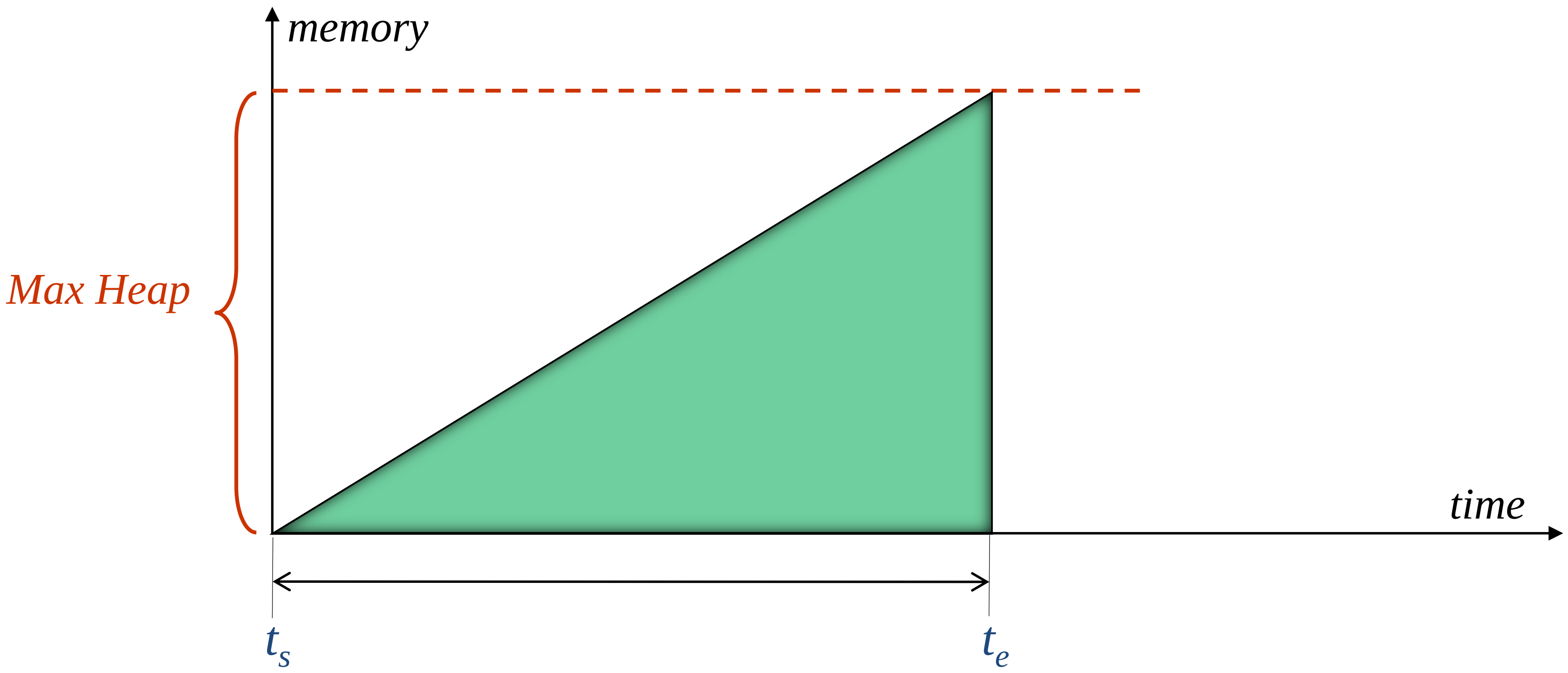
Execution Time vs. Memory



Execution Time vs. Memory



Execution Time vs. Memory



When to collect

- “Stop-the-world”
  - All mutators stop while collection is ongoing
- Incremental
  - Mutators perform small chunks of marking during each allocation
- Concurrent/Parallel
  - Garbage collection happens in concurrently running thread – requires some kind of synchronization between mutator & collector

Programmer’s Perspective

- Your program is running out of memory. What do you do?
- Possible reasons:
  - Leak
  - Bloat
- Your program is running slowly and unpredictably
  - Churn
  - “GC Thrashing”

Memory Leaks

- Objects that remain reachable, but will not be accessed in the future
  - Due to application semantics
- Will ultimately lead to out-of-memory condition
  - But will degrade performance before that
- Common problem, particularly in multi-layer frameworks
  - Containers are a frequent culprit
  - Heap profilers can help

Bloat and Churn

- Bloat: use of inefficient, pointer-intensive data structures increases overall memory consumption [Chis et al 2011]
  - E.g. HashMaps for small objects
- Churn: frequent and avoidable allocation of objects that turn to garbage quickly

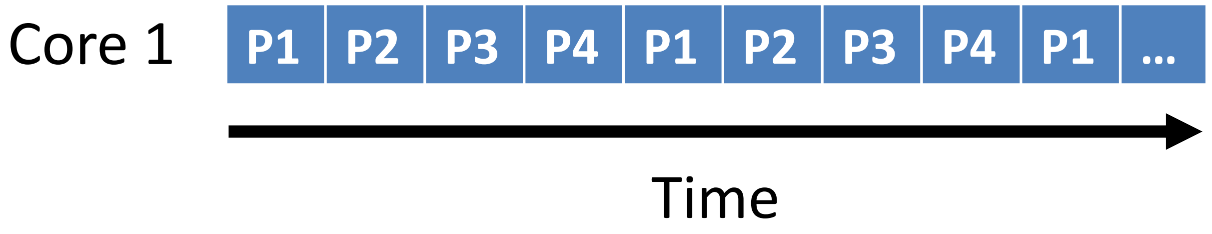
CONCURRENCY

Concurrent/Parallel Programs

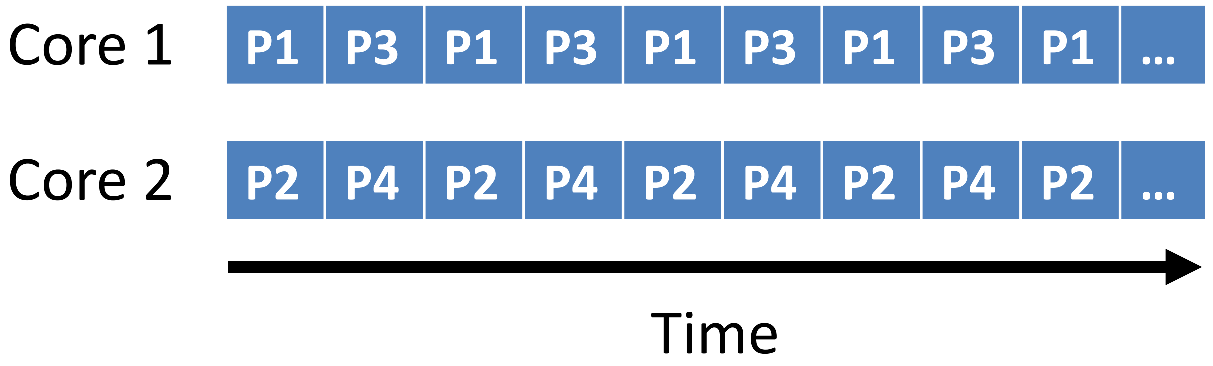
- To execute these programs we need to
  - **Create several processes** that execute in parallel (or concurrently)
  - Map each process to the **same address space to share data**
    - They are all part of the same computation
  - Have the **OS schedule** these processes in parallel
- This situation is very inefficient. Why?

Concurrency vs. Parallelism

- Concurrent execution on a single-core system:



- Parallel execution on a dual-core system:



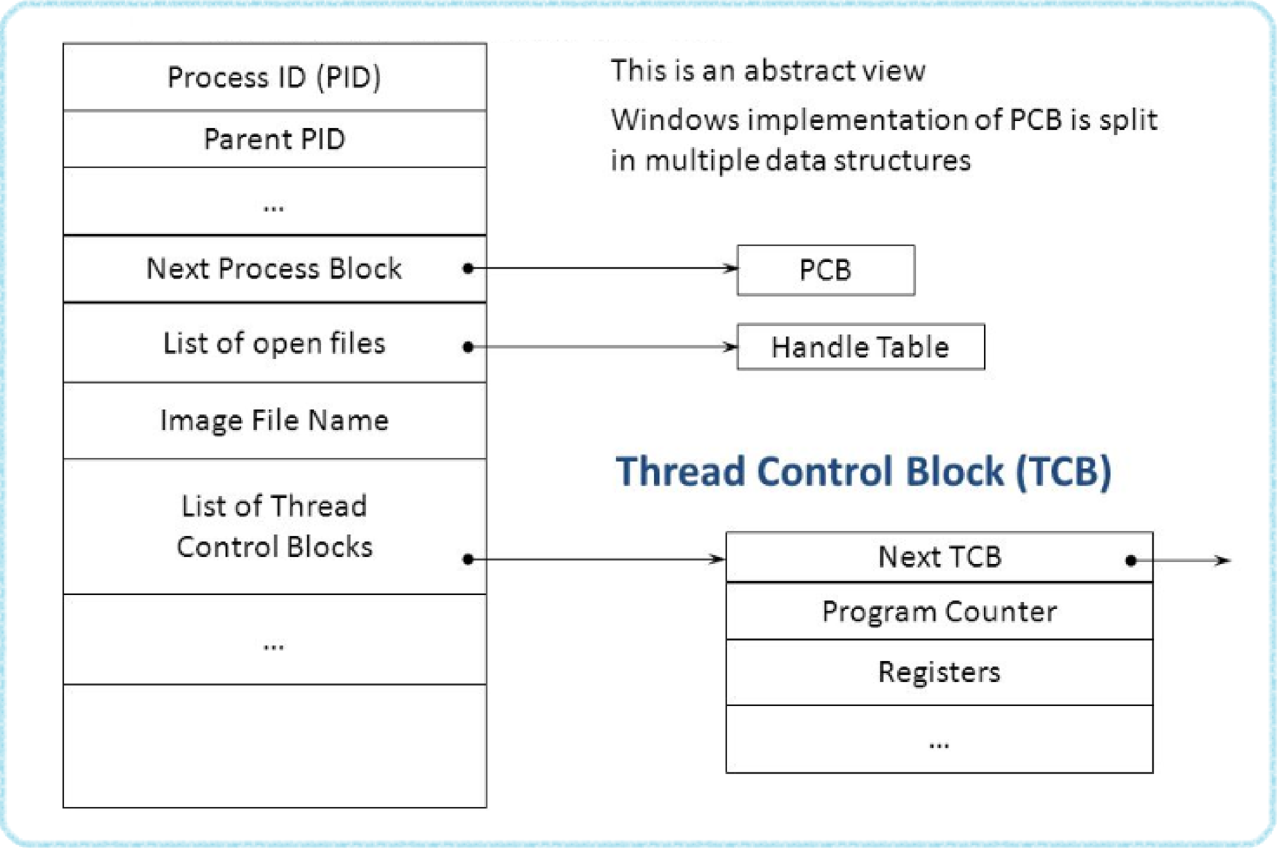
Thread

- A **thread** is a single execution sequence that represents a separately schedulable task

- Multi-threaded program
  - A multi-threaded program has **more than one point of execution**.
  - Multiple PCs (Program Counter)
  - They share the share the same address space.

Thread Control Block (TCB)

- Thread ID
- Thread state
- Pointer to parent PCB
- PC/registers for thread
- Stack location in memory map.



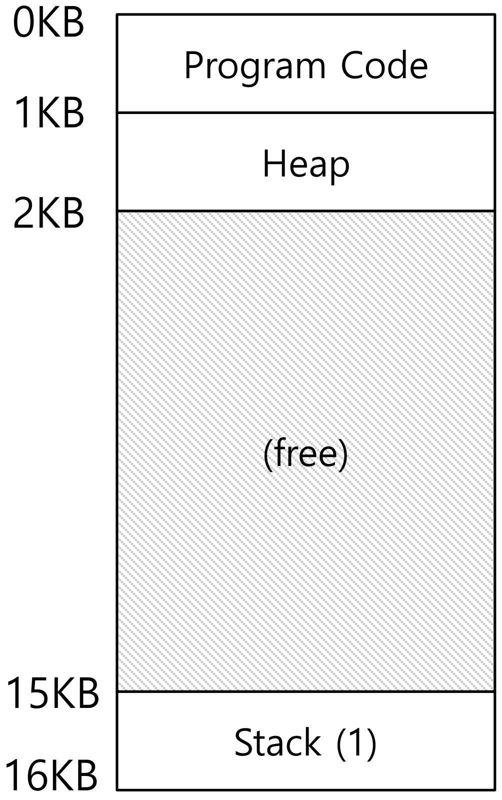
Context switch between threads

- 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.

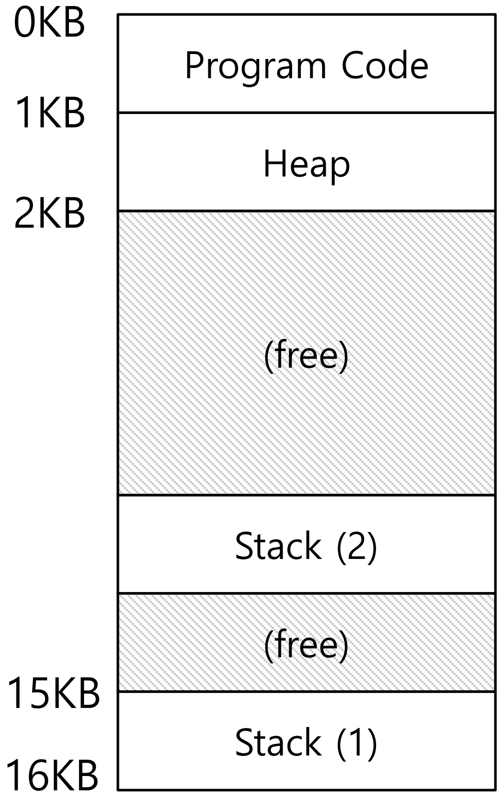


The code segment: where instructions live

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

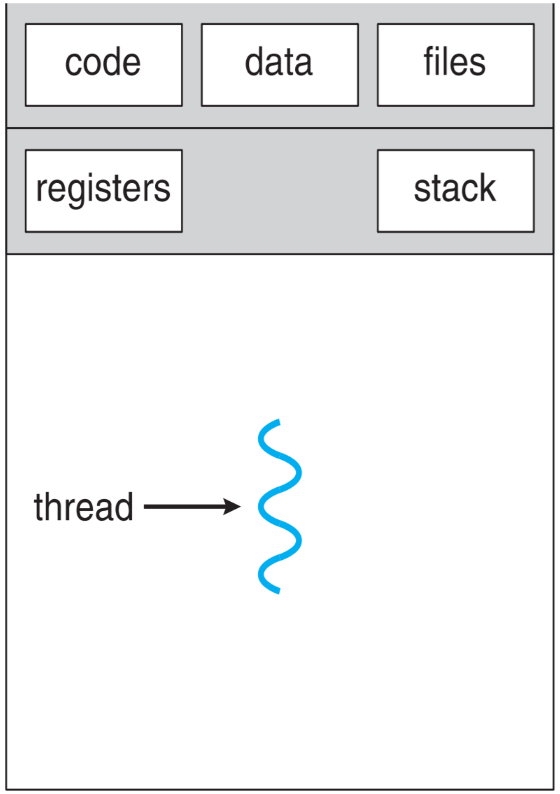
(it grows upward)
The stack segment: contains local variables arguments to routines, return values, etc.

A Single-Threaded Address Space

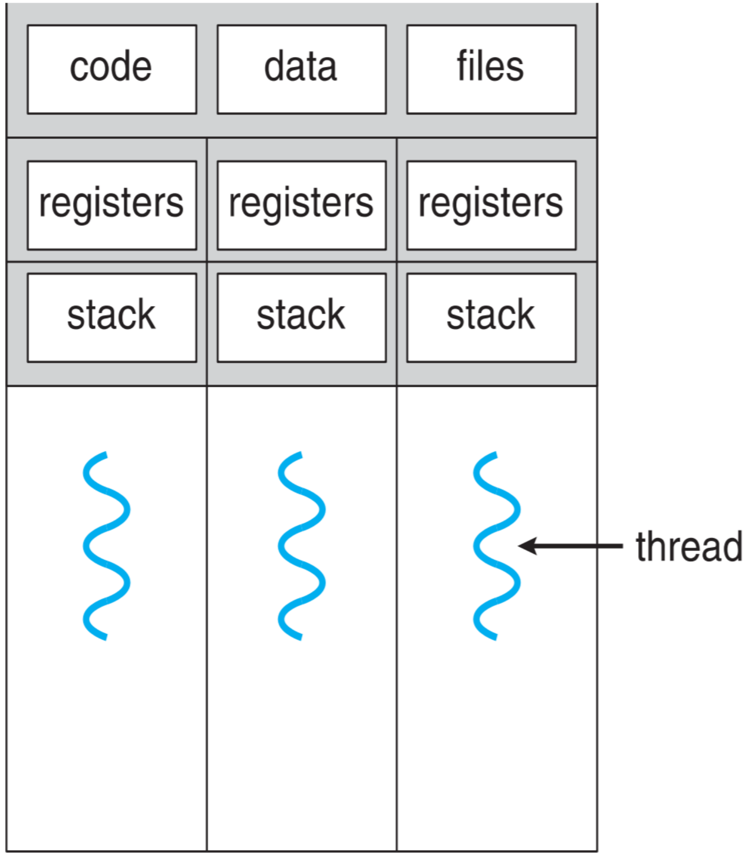


Two threaded Address Space

Single vs. Multithreaded Processes



single-threaded process



multithreaded process

Why Use Threads?

- Parallelism
  - One thread per CPU can make programs run faster on multiple processors
- I/O overlapping
  - Avoid blocking program progress due to slow I/O
  - While one thread in your program waits (i.e., blocked waiting for I/O), the CPU scheduler can switch to other threads, which are ready to run and do something useful.
  - Similar to the effect of multiprogramming

Why Use Threads?

- You could use multiple processes instead of threads.
  – Processes are a more sound choice for logically separate tasks
- But,
  – Process creation is heavy-weight while thread creation is lightweight
  – Threads share an address space and thus make it easy to share data
  – Can simplify code, increase efficiency

Threads are “lighter weight” than processes

- To make a new thread, we only need a **stack.**
  - Can share all pre-existing resources.
  - Done at user-level without system calls
- To make a new process, we have to talk to the **operating system**
  - Make a new address space
  - Inherit resources from the original space
  - Compete for the same underlying global scheduler
  - Can easily run out, or thrash

Benefits

- Responsiveness
  – may allow continued execution if part of process is blocked, especially important for user interfaces
- Resource Sharing
  – threads share resources of process, easier than shared memory or message passing
- Economy
  – cheaper than process creation, thread switching lower overhead than context switching
- Scalability
  – Threads can take advantage of multiprocessor architectures

An Example: Thread Creation

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread (void *arg) {
    printf (“%s\n”, (char *) arg);
    return NULL;
}

int main (int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf(“main: begin\n”);
    pthread_create(&p1, NULL, mythread, “A”);
    pthread_create(&p2, NULL, mythread, “B”);
    // join waits for thre threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf(“main: end\n”);
    return 0;
}
```

Thread Trace (1)

main	**Thread 1**	**Thread 2**
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
		runs
		prints “B”
		returns
prints “main: end”		

Thread Trace (2)

main	Thread 1	Thread 2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Thread Trace (3)

main	**Thread 1**	**Thread 2**
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2 returns immediately; T2 is done		
prints "main: end"		

Issues when using threads

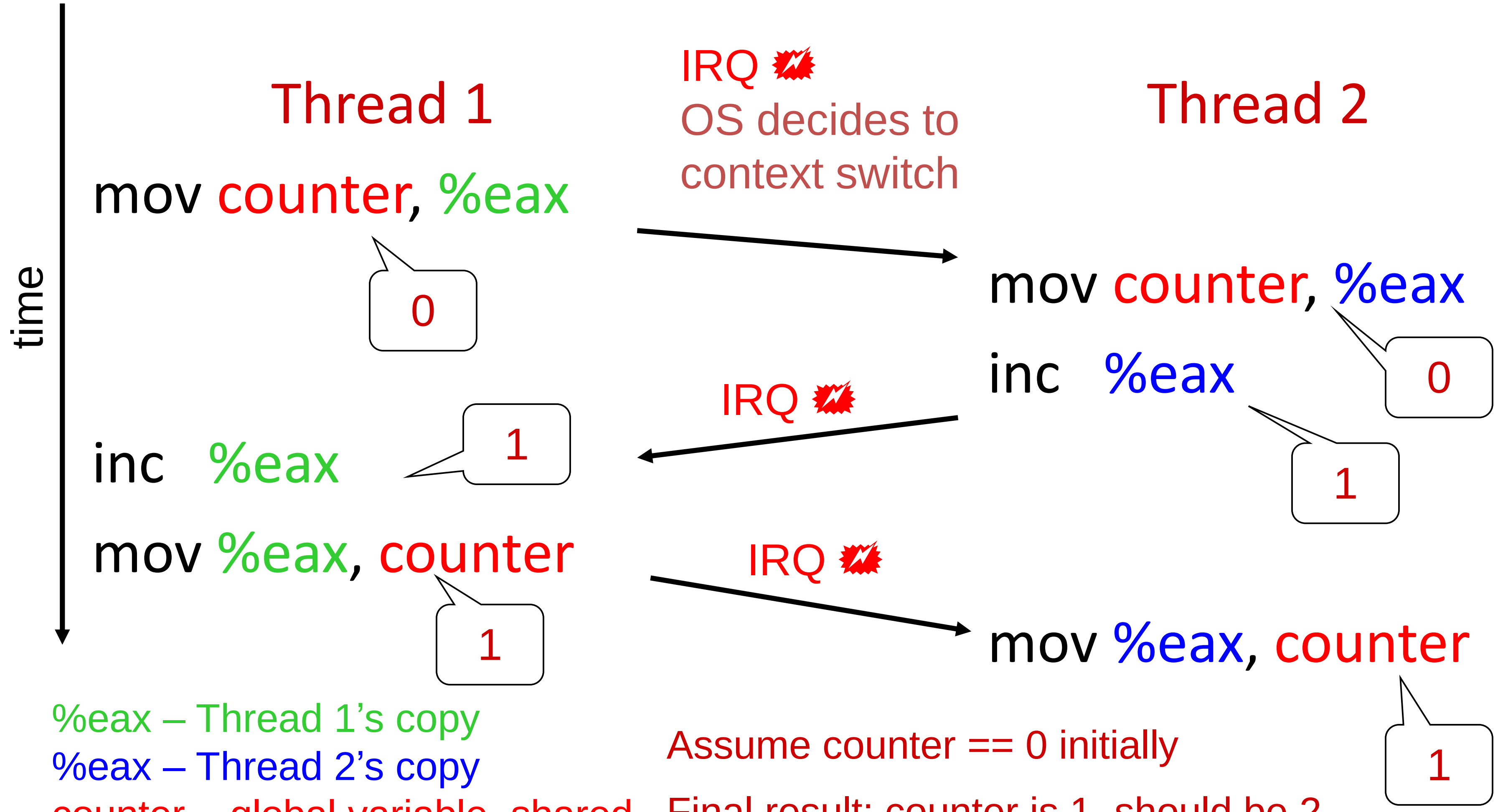
```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>jjj
int counter = 0;
void *mythread (void *arg) {
    printf (“%s\n”, (char *) arg);
    for(int i = 0; i < 1e7; i++)
       counter = counter + 1;
    printf (“%s: done\n”, (char *) arg);
    return NULL;
}

int main (int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf(“main: begin\n”);
    pthread_create(&p1, NULL, mythread, “A”);
    pthread_create(&p2, NULL, mythread, “B”);
    // join waits for thre threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf(“main: end\n”);
    return 0;
}
```

Result

```
main: begin (counter = 0)
A
B
B: done
A: done
main: end (counter = 10313459)
```

```
main: begin (counter = 0)
A
B
A: done
B: done
main: end (counter = 10201521)
```

Race Conditions



%eax – Thread 1's copy
%eax – Thread 2's copy
counter – global variable, shared

Assume counter == 0 initially

Final result: counter is 1, should be 2

Race Conditions

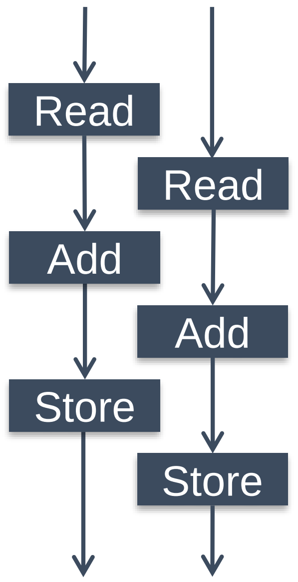
- Race condition
  - Two threads “race” to execute code and update shared (dependent) data
  - Errors emerge based on the ordering of operations, and the scheduling of threads
  - Thus, errors are nondeterministic

Critical Sections

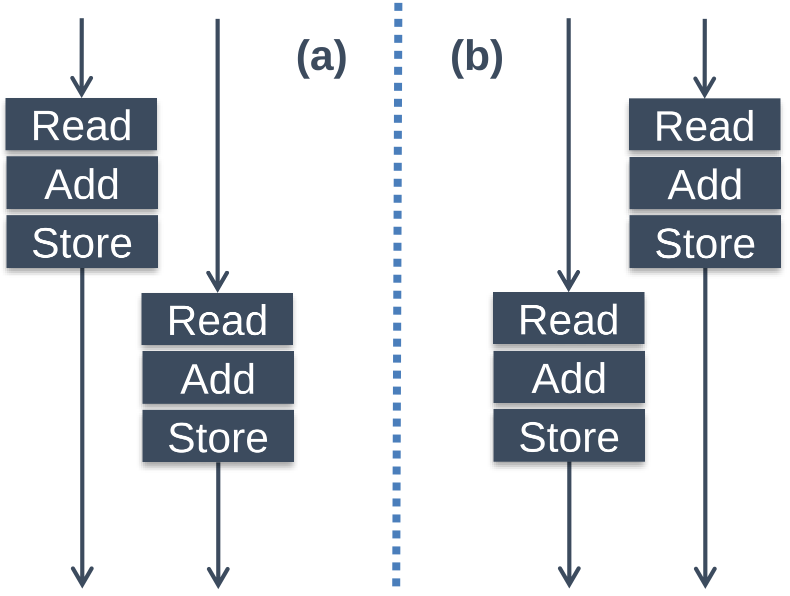
- Classical definition of a critical section:
  - *"A piece of code that* ***accesses a shared resource*** *that must not be concurrently accessed by more than one thread of execution."*
  - Multiple threads executing critical section can result in a race condition.
  - Need to support **atomicity** for critical sections (**mutual exclusion**)
- Two problems
  - Code was not designed for concurrency
  - Shared resource (data) does not support concurrent access

Wish for Atomicity

- We need more powerful instructions
  – do exactly whatever we needed done in a single step → atomic
  – remove the possibility of an untimely interrupt



Interleaved Execution



Non-Interleaved (Atomic) Execution

One More Problem: Waiting For Another

- Another common interaction
  - One thread must wait for another to complete some action before it continues
  - e.g., when a process performs a disk I/O and is put to sleep; when the I/O completes, the process needs to be roused from its slumber so it can continue

Thread Lifecycle