Universidade Estadual de Campinas
Instituto de Computação
MC504 Sistemas Operacionais

T16 Concurrency and Threads

Referência principal
Ch.26 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 17 de setembro de 2018

Arthur João Catto, PhD

2º semestre de 2018

Introduction to Concurrency

- We use the word concurrency to refer to multiple activities that can happen at the same time.
- Correctly managing concurrency is a key challenge for operating system developers and, more recently, has also become a concern for many application developers.
- From the programmer's perspective, it is much easier to think sequentially than to keep track of many simultaneous activities.

How can you write a correct program with dozens of events happening at once?

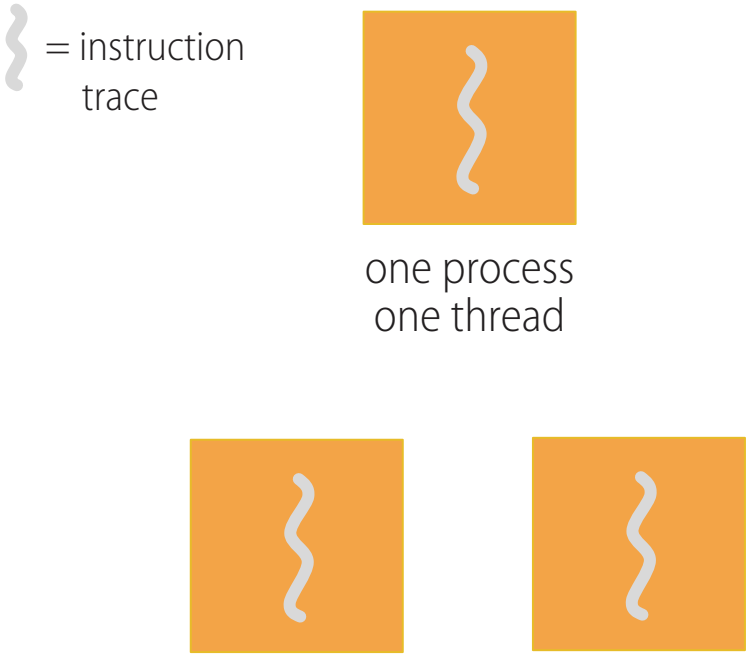
How do you debug such program if each execution is unique?

Concurrency within Applications

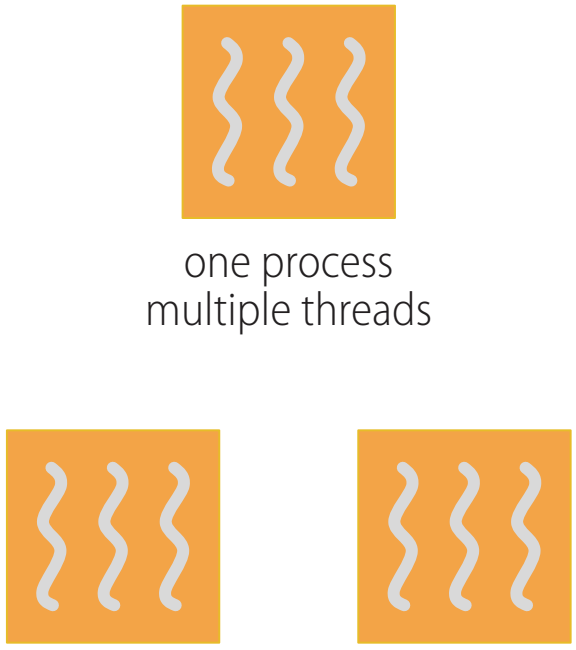
- We have already studied two abstractions
  - **Virtual CPUs** – enabling concurrency between multiple running programs (i.e. processes).
  - **Virtual Memory** – defining **address spaces** which enable each program to feel like having its own private memory.
- We will now introduce a new abstraction: **threads**
  - Threads behave nearly as separate processes, except for **one big difference**:
    - All the threads of a program share the same address space.
  - Threads enable concurrency within a single process.
  - A **multi-threaded program** will have more than one point of execution but only a single address space.

Processes and threads

- A traditional (heavyweight) process has a single thread of control



- A multi-threaded process can have more than one active task at a time



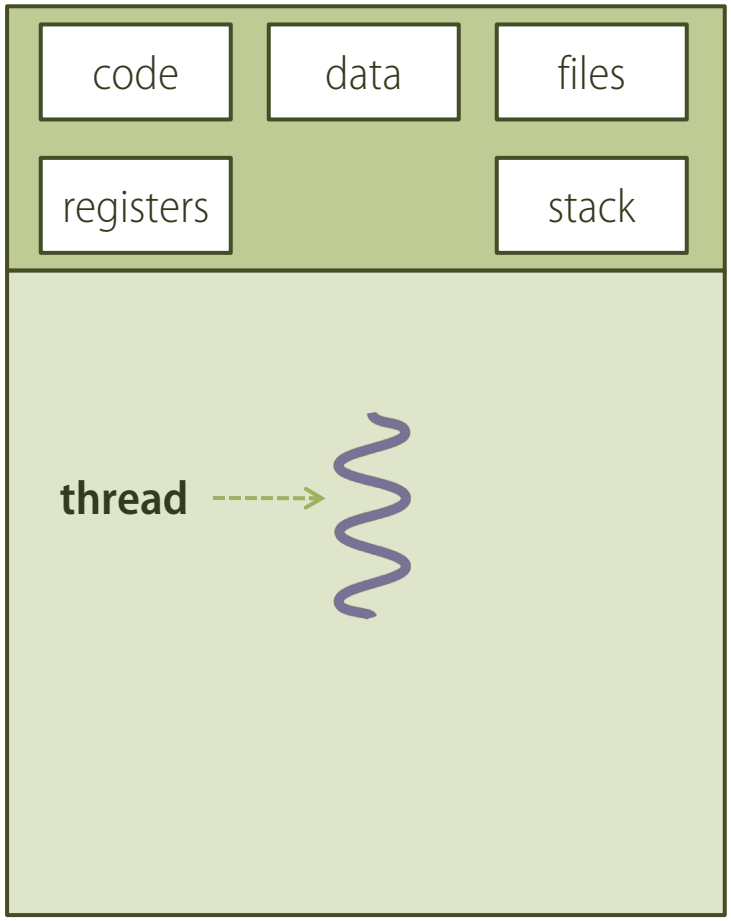
What's our goal?

- The key idea is to write a concurrent program as a set of sequential streams of execution, or **threads**, that interact and share results in a very precise way.
- **Threads** let us define a set of **sequential tasks that run concurrently**.
- We will now
  - define the thread abstraction
  - show how you can use it
  - explain how the OS can implement threads on top of a limited number of processors.

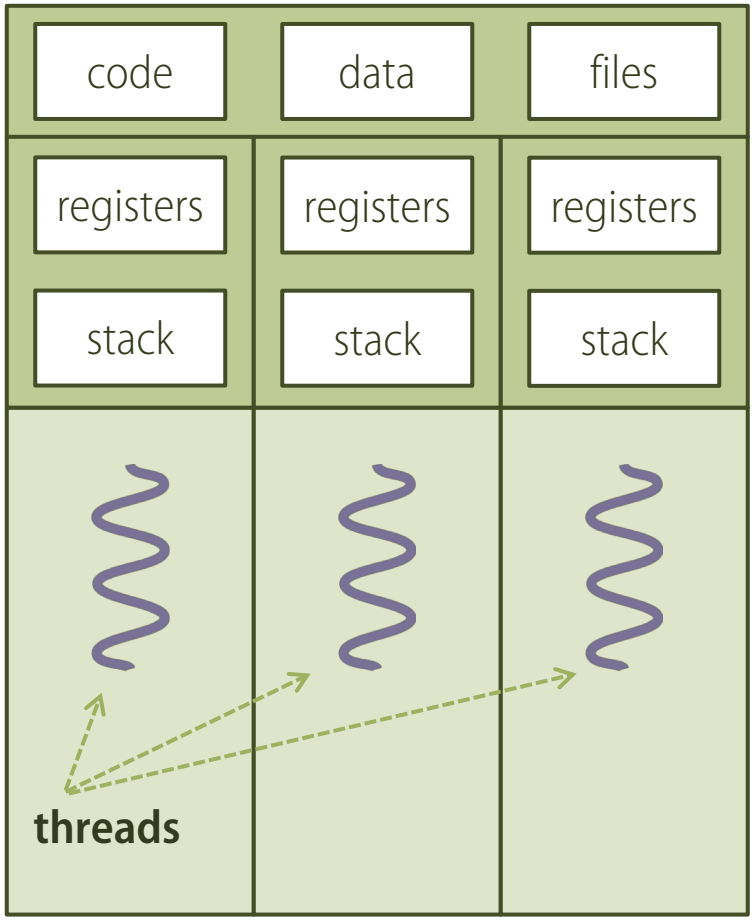
Thread Control Blocks (TCBs)

- Each thread has its own private set of registers it uses for computation
- If there are two threads that are running on a single processor, switching from running one to running the other, requires a context switch.
- The context switch between threads is quite similar to the context switch between processes, as the register state of the first must be saved and the register state of the second restored before running it.
- For processes, we saved state to a Process Control Block (PCB).
  For threads, we'll need one or more Thread Control Blocks (TCBs).

A process may have many threads

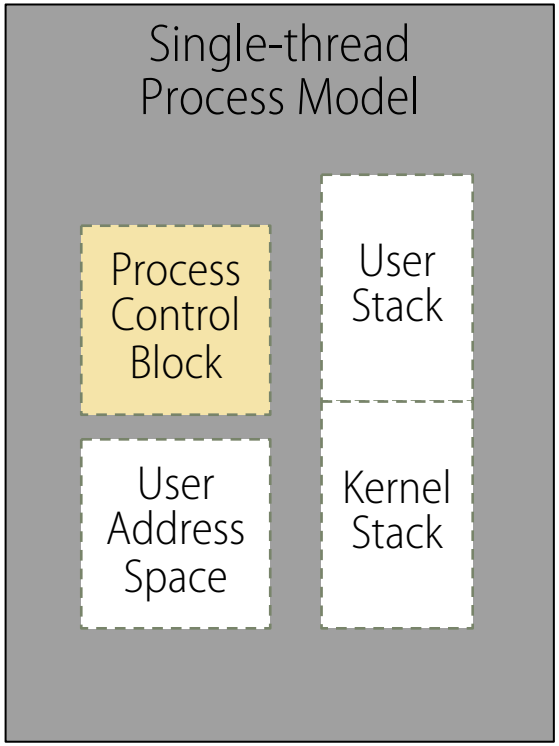


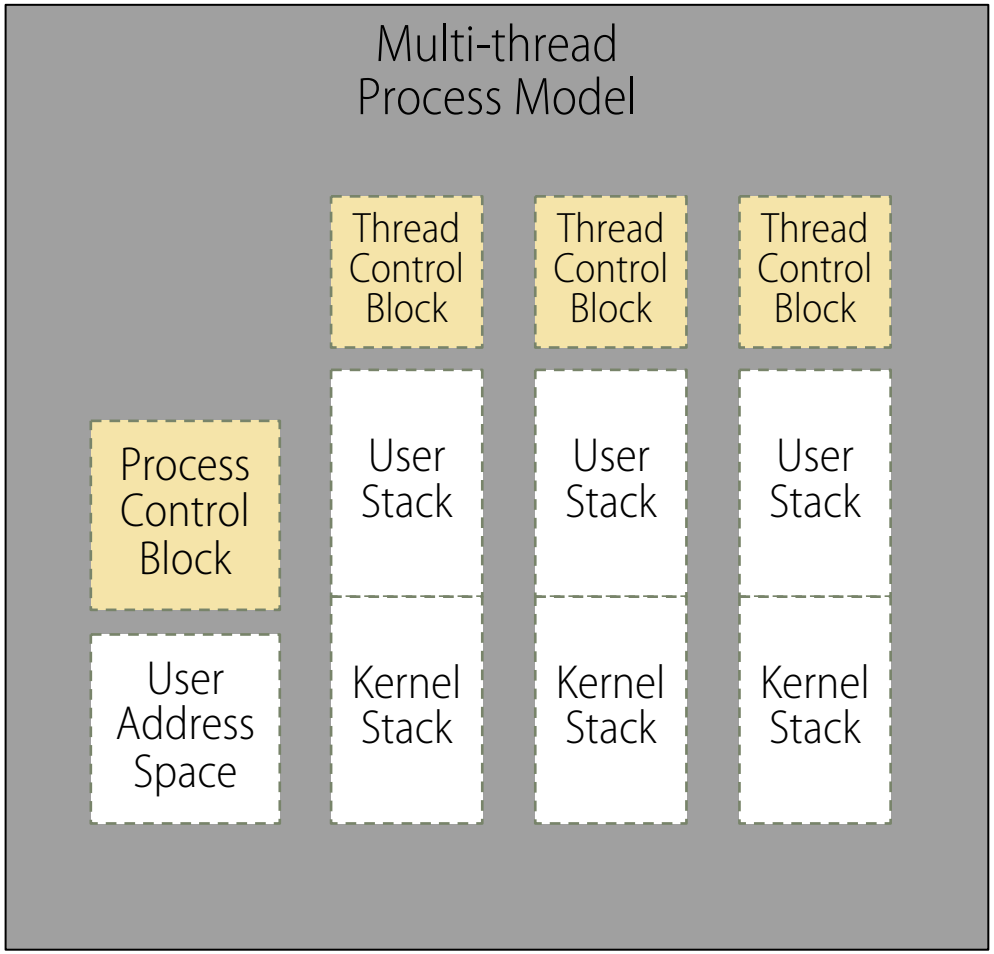
Single-threaded process



Multi-threaded process

Single-thread and multi-thread process models





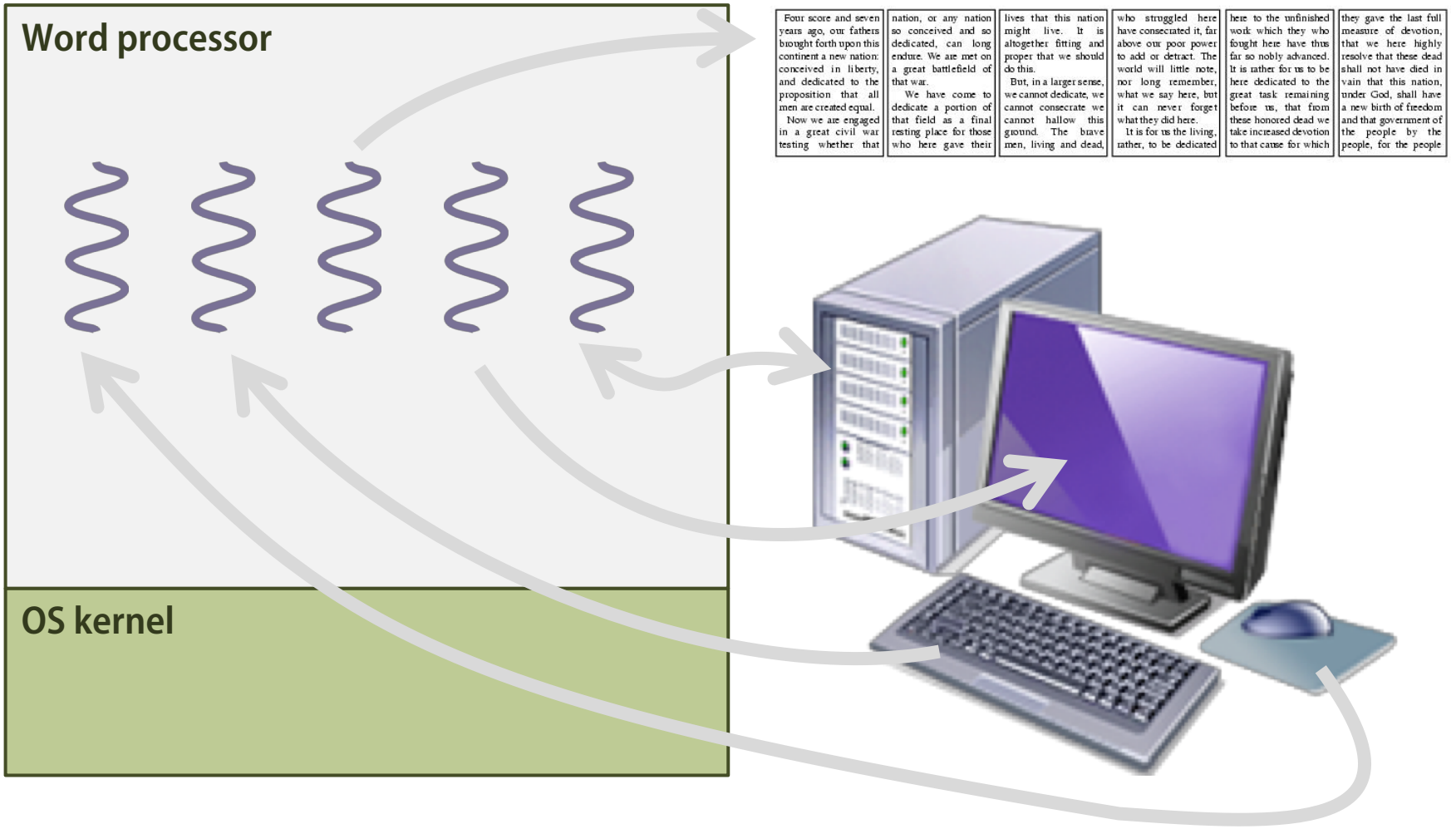
Examples of process and thread level items

Per process items (shared by all threads in a process)	Per thread items (private to each thread)
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Processes and threads

- An OS can be seen as a manager of
  - resources
  - time
- Early OSs treated both problems at process level.
- Modern OSs use a finer grain for time management:
  - The unit of **dispatching** is now a thread or lightweight process.
  - The unit of **resource** ownership still is a process or task.

A multithreaded word processor



A thread is a single execution sequence that represents a separately executable task.

Single execution sequence

- Each thread executes a sequence of instructions just as in the familiar sequential programming model.

Separately executable task

- The operating system can run, suspend or resume a thread at any time.

In summary, a thread . . .

- Has an execution state.
  - Running, ready, etc.
- Has an execution stack.
- Has some static storage for local variables.
- Has its context saved when not running.
- Has access to the memory and resources of its process.
  - These are shared by all threads of the process
- Is blocked when its process is blocked.
- Is terminated when its process terminates.

Why should you use threads at all?

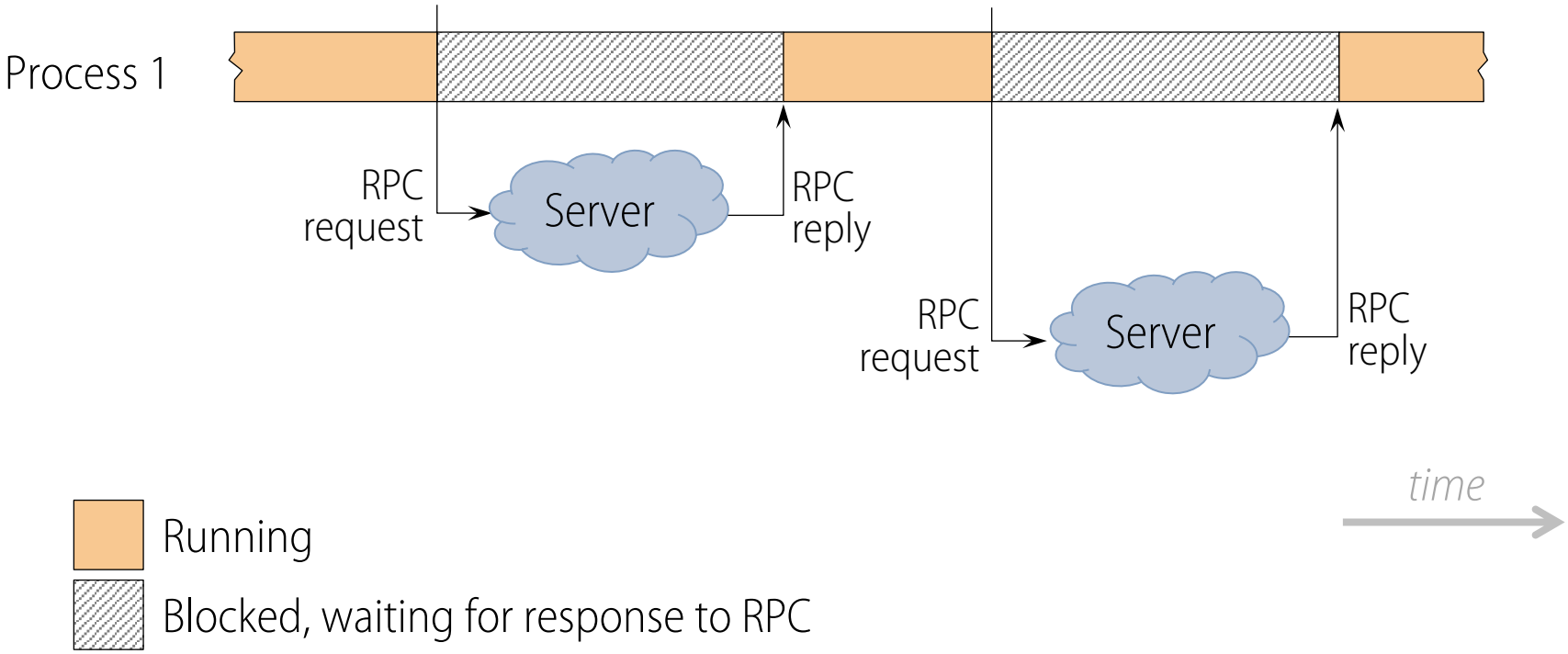
Exploiting parallelism	• E.g. by doing foreground and background work at the same time.
Asynchronous processing	• E.g. by implementing asynchronous elements of a program (power failure protection, backup, spell check, etc.) as independent threads.
Speeding execution	• E.g. by overlapping record reading and processing.
Modular program structure	• Programs involving different activities or data sources and destinations may be easier to design and implement using separate threads.

Thread benefits

Responsiveness	• A multithreaded process may run even if part of it is blocked or very busy.
Resource sharing	• The threads of a process share memory and files, so intercommunication may not require kernel involvement.
Performance	• It is faster to create a new thread than a process, to terminate a thread than a process and to switch between threads than between processes.
Scalability	• A multi-threaded process can benefit of a multiprocessor or multicore architecture.

Example

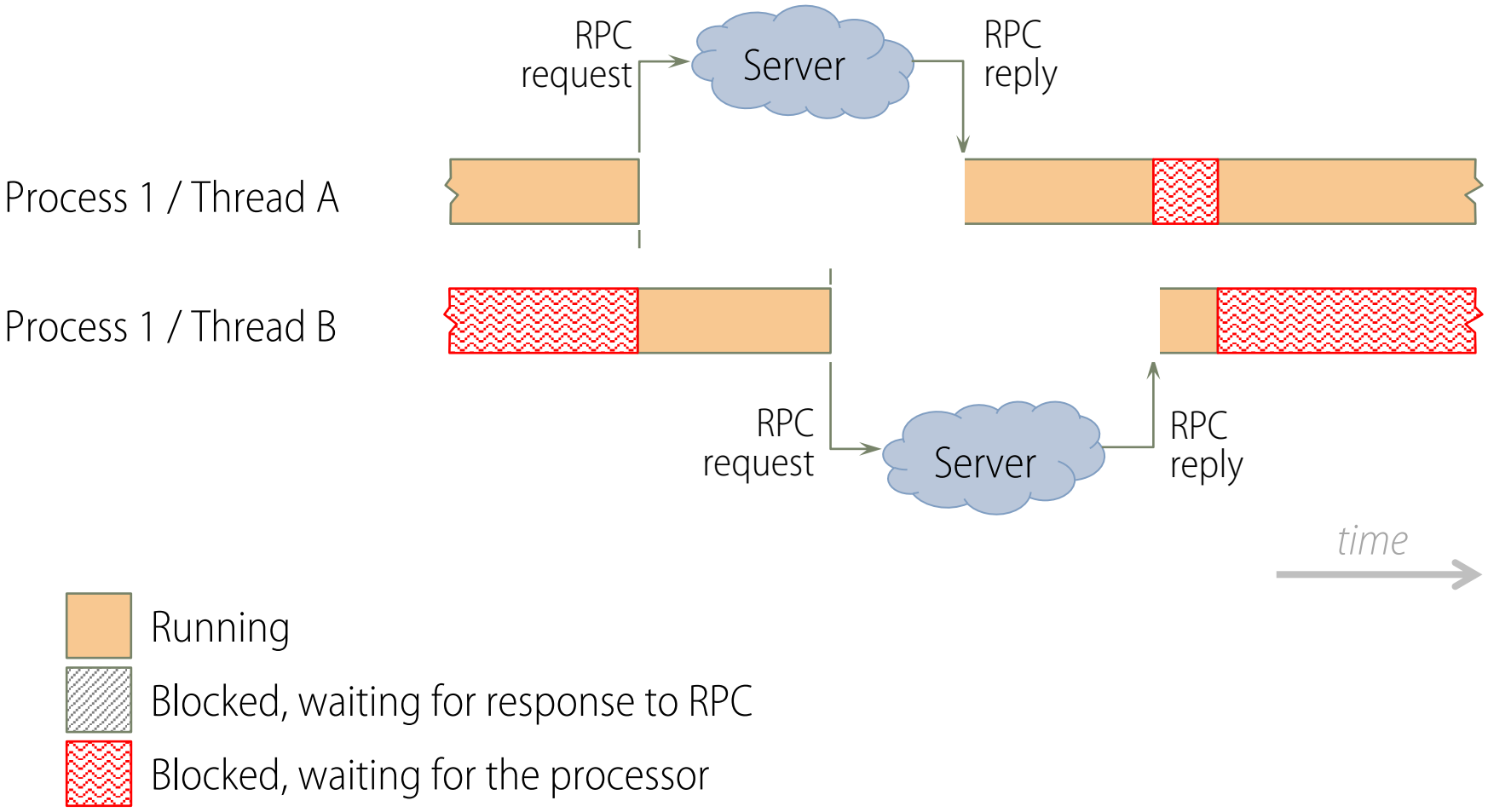
Synchronous RPC in a single thread uniprocessor



RPC ≡ Remote Procedure Call

Example

Synchronous RPC in a multi thread uniprocessor



Multicore programming challenges

Dividing activities	• Finding program blocks that can be run in parallel.
Achieving balance	• Adjusting block run duration and value.
Reducing data dependency	• Minimizing data dependencies between blocks. • Assuring proper synchronization of the unavoidable dependencies.
Testing and debugging	• Process traces are of much lower value than in single-core or single-processor environments due to potential nondeterminism

Example: Thread creation

- Let's design a 2-thread application so that one of the threads will print "A" while the other thread will print "B".
- We will use two functions from the **pthread** API of POSIX
  - Create a new thread

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *func, void *arg);
```

  - Wait for the termination of another thread

```
int pthread_join(pthread_t thread, void **thread_return);
```

- The POSIX API provides a way to invoke an asynchronous procedure call.
  - An asynchronous procedure call separates the call from the return.
    - The caller starts the function and continues executing concurrently with the called function.
    - Later, the caller can wait for function completion.

Before we go . . . let's wrap the pthread functions

- If you examine pthread's specification, you'll see that some of its functions return a success indicator.
- In order to avoid processing the success indicator at each point of call, we'll define wrapper functions, which will just call their similarly-named counterparts and make sure that they don't return any funny results.
- The "`mythreads.h`" header file will contain our wrapper functions, which differ from the original ones by having their names starting with a capital "P".

Our wrapper functions

```
1   #ifndef __MYTHREADS_h__
2   #define __MYTHREADS_h__
3
4   #include <pthread.h>
5   #include <assert.h>
6   #include <sched.h>
7
8   void Pthread_create(pthread_t *thread, const pthread_attr_t *attr,
9                       void *(*start_routine)(void*), void *arg) {
10      int rc = pthread_create(thread, attr, start_routine, arg);
11      assert(rc == 0);
12  }
13
14  void Pthread_join(pthread_t thread, void **value_ptr) {
15      int rc = pthread_join(thread, value_ptr);
16      assert(rc == 0);
17  }
18
19  #endif // __MYTHREADS_h__
```

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "mythreads.h"

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

```

```
10
11  int main(int argc, char *argv[]) {
12
13      pthread_t p1, p2;
14      printf("main : begin\n");
15      Pthread_create(&p1, NULL, mythread, "A");
16      Pthread_create(&p2, NULL, mythread, "B");
17
18      // join waits for the threads to finish
19      Pthread_join(p1, NULL);
20      Pthread_join(p2, NULL);
21
22      printf("main : end\n");
23      return 0;
24  }
```

```
SUP080:code arthur.catto$ gcc -o p0 p0.c -Wall -pthread
SUP080:code arthur.catto$ ./p0
main : begin
A
B
main : end
SUP080:code arthur.catto$
```

Is this the only possible output we could get?

Example(?)

Making things worse: sharing data

Design a 2-thread application so that

- Each thread performs the summation of the sequence of non-negative integers up to a given limit.
- The main function prints the accumulated result of the threads' calculation.

```c
#include <stdio.h>
#include "mythreads.h"
#include <stdlib.h>
#include <pthread.h>

int max;
volatile int counter = 0; // shared global variable

void *mythread(void *arg) {
    char *letter = arg;
    int i;              // stack (private per thread)
    printf("%s: begin [addr of i: %p]\n", letter, &i);
    for (i = 0; i < max; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

```c
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: main-first <loopcount>\n");
        exit(1);
    }
    max = atoi(argv[1]);

    pthread_t p1, p2;
    printf("main: begin [counter = %d] [%x]\n", counter,
           (unsigned int) &counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [counter: %8d]\n [should:  %8d]\n",
           counter, max*2);
    return 0;
}
```

```
SUP080:nb180917 arthur.catto$ gcc -o t1 t1.c -Wall -pthread
SUP080:nb180917 arthur.catto$ ./t1 10000000
main: begin [counter = 0] [dc2f060]
A: begin [addr of i: 0x700010038efc]
B: begin [addr of i: 0x7000100bbefc]
B: done
A: done
main: done
 [counter: 10455431]
 [should:  20000000]
SUP080:nb180917 arthur.catto$ ./t1 10000000
main: begin [counter = 0] [5c5060]
A: begin [addr of i: 0x700000293efc]
B: begin [addr of i: 0x700000316efc]
B: done
A: done
main: done
 [counter: 10338269]
 [should:  20000000]
SUP080:nb180917 arthur.catto$
```

Where is the bug?

Example: Summing a range of integers

Design a proper multithreaded application that performs the summation of the sequence of non-negative integers up to a given limit.

- Main idea
  - Split the sequence into nt ranges.
  - Create nt threads and let each of them sum one range.
  - Sum the results of all threads.

```c
#include <stdio.h>
#include "mythreads.h"
#include <pthread.h>
#include <stdlib.h>

#define N 100
#define NUM_THREADS 6

void *adder(void *param); /* the thread */

int main(void) {
    pthread_t tid[NUM_THREADS];                    /* thread ids */
    int thread_arg[NUM_THREADS][4];
    int sum = 0;
    int range = N / NUM_THREADS;

    /* create the threads */
```

...

```c

    /* wait for the threads to exit and calculate total */
```

...

```c

    printf("Main again... sum = %d.\n", sum);
}

```

```
16
17      /* create the threads */
18      for (int i = 0; i < NUM_THREADS; ++i) {
19          thread_arg[i][0] = i;
20          thread_arg[i][1] = i * range + 1;
21          thread_arg[i][2] = (i == NUM_THREADS-1) ? N : (i+1) * range;
22          printf("Main creating thread %d to sum %3d to %3d.\n",  i,
 •          thread_arg[i][1], thread_arg[i][2]);
23          Pthread_create(&tid[i], NULL, adder,  (void *) &thread_arg[i][0]);
24      }
25
```

```
25
26      /* wait for the threads to exit and calculate total */
27      for (int i = 0; i < NUM_THREADS; ++i) {
28          Pthread_join(tid[i], NULL);
29          printf("Thread %d returned %d.\n", i, thread_arg[i][3]);
30          sum += thread_arg[i][3];
31      }
32
```

```
35
36  void *adder(void *param) {
37      int i;
38      int ix    = *((int *)param);
39      int lower = *((int *)param + 1);
40      int upper = *((int *)param + 2);
41
42      printf("Thread %d summing %3d to %3d...\n",  ix, lower, upper);
43
44      int sum = 0;
45
46      for (i = lower; i <= upper; i++)
47          sum += i;
48
49      *((int *)param + 3) = sum;
50      printf("Thread %d summed %3d to %3d... Result %d.\n",  ix, lower, upper,
 •      sum);
51
52      pthread_exit(NULL);
53  }
```

Example 3

Sample result output

```
SUP080:nb180917 arthur.catto$ gcc -o t2 t2.c -Wall -pthread
SUP080:nb180917 arthur.catto$ ./t2
Main creating thread 0 to sum   1 to  20.
Main creating thread 1 to sum  21 to  40.
Main creating thread 2 to sum  41 to  60.
Thread 0 summing   1 to  20...
Thread 1 summing  21 to  40...
Main creating thread 3 to sum  61 to  80.
Thread 2 summing  41 to  60...
Thread 3 summing  61 to  80...
Thread 2 summed  41 to  60... Result 1010.
Thread 0 summed   1 to  20... Result 210.
Main creating thread 4 to sum  81 to 100.
Thread 3 summed  61 to  80... Result 1410.
Thread 1 summed  21 to  40... Result 610.
Thread 4 summing  81 to 100...
Thread 4 summed  81 to 100... Result 1810.
Thread 0 returned 210.
Thread 1 returned 610.
Thread 2 returned 1010.
Thread 3 returned 1410.
Thread 4 returned 1810.
Main again... sum = 5050.
SUP080:nb180917 arthur.catto$
```

Pthread examples

A few functions of the POSIX pthread library

- Create a new thread
  - `int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *func, void *arg);`
- Terminate the calling thread
  - `void pthread_exit(void *retval);`
- Wait for termination of another thread
  - `int pthread_join(pthread_t thread, void **thread_return);`
- Thread cancellation
  - `int pthread_cancel(pthread_t thread);`

Example 1

Design a 2-thread application so that

- One of the threads prints “A” and
- The other thread prints “B”.

```c
#include <stdio.h>
#include "mythreads.h"
#include <stdlib.h>
#include <pthread.h>

```

```c

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

```

```c
#ifndef __MYTHREADS_h__
#define __MYTHREADS_h__

#include <pthread.h>
#include <assert.h>
#include <sched.h>

void Pthread_mutex_lock(pthread_mutex_t *m) {
    int rc = pthread_mutex_lock(m);
    assert(rc == 0);
}

void Pthread_mutex_unlock(pthread_mutex_t *m) {
    int rc = pthread_mutex_unlock(m);
    assert(rc == 0);
}

void Pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                    void *(*start_routine)(void*), void *arg) {
    int rc = pthread_create(thread, attr, start_routine, arg);
    assert(rc == 0);
}

void Pthread_join(pthread_t thread, void **value_ptr) {
    int rc = pthread_join(thread, value_ptr);
    assert(rc == 0);
}

#endif // __MYTHREADS_h__
```

```c

int main(int argc, char *argv[]) {
    if (argc != 1) {
        fprintf(stderr, "usage: main\n");
        exit(1);
    }

    pthread_t p1, p2;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

```
SUP080:nb180917 arthur.catto$ gcc -o t0 t0.c -Wall -pthread
SUP080:nb180917 arthur.catto$ ./t0
main: begin
A
B
main: end
SUP080:nb180917 arthur.catto$
```

Could the output be any different?

Example 2

Design a multithreaded application that shows that threads may continue after the end of main.

Example 2

General solution layout

```
1.  #include <time.h>
2.  #include <pthread.h>
3.  #include <stdio.h>
4.  #include <stdlib.h>
5.  #include <assert.h>

6.  #define NUMBER_OF_THREADS 10

7.  void *runner(void *tix) {
8.     /* Print thread's index and busy wait for some time */
9.     ...
10. }

11. int main(int argc, char *argv[]) {
12.    /* create NUMBER_OF_THREADS threads and exit */
13.    ...
14. }
```

Example 2

Thread prints start time, busy waits and exits

```
1.  void *runner(void *tix) {
2.      /* Print thread's index and busy wait for some time */
3.      clock_t startTime, endTime;
4.      double totalTime;
5.
6.      startTime = clock();
7.      printf("%6ld  Thread %d starting.\n", startTime, *((int *)tix));
8.
9.      for(int i = 0; i < 10000000; ++i) { }
10.
11.     endTime = clock();
12.     totalTime = (double) (endTime - startTime) / CLOCKS_PER_SEC;
13.     printf("%6ld  Exiting thread %d after %.2lfs.\n", endTime, *((int *)tix), totalTime);
14.     pthread_exit(NULL);
15. }
```

Example 2

Main creates threads and terminates

```
1.  int main(int argc, char *argv[]) {
2.     pthread_t threads[NUMBER_OF_THREADS];
3.     int threadIx[NUMBER_OF_THREADS];
4.     int status, i;
5.     clock_t startTime, endTime, threadTime;
6.     double totalTime;
7.
8.     startTime = clock();
9.     printf("%ld  Main starting.\n", startTime);
10.
11.    /* create threads */
12.
13.    endTime = clock();
14.    totalTime = (double) (endTime - startTime) / CLOCKS_PER_SEC;
15.    printf("%6ld  Main exiting after %.2lfs.\n", endTime, totalTime);
16.    pthread_exit(NULL);
17. }
```

Example 2

Thread creation in main

```
1.      /* create the threads */

2.    for(i = 0; i < NUMBER_OF_THREADS; i++) {
3.       threadTime = clock();
4.       printf("%6ld  Creating thread %d.\n", threadTime, i);
5.       threadIx[i] = i;
6.       status = pthread_create(&threads[i], NULL, runner,
                                 (void *) &threadIx[i]);
7.       assert(status == 0);
8.    }
```

Example 2

Sample output result

```
1387  Main starting.
1417  Creating thread 0.
1456  Creating thread 1.
1477  Creating thread 2.
1487  Thread 0 starting.
1501  Creating thread 3.
1516  Thread 2 starting.
1517  Thread 1 starting.
1583  Creating thread 4.
1600  Thread 3 starting.
1658  Creating thread 5.
1664  Thread 4 starting.
1696  Creating thread 6.
1705  Thread 5 starting.
1812  Creating thread 7.
2141  Creating thread 8.
2216  Creating thread 9.
2270  Main exiting after 0.00s.
3822  Thread 6 starting.
3916  Thread 7 starting.
43760  Thread 8 starting.
63200  Thread 9 starting.
173646  Exiting thread 6 after 0.17s.
187023  Exiting thread 1 after 0.19s.
200164  Exiting thread 7 after 0.20s.
203476  Exiting thread 4 after 0.20s.
215335  Exiting thread 2 after 0.21s.
220500  Exiting thread 5 after 0.22s.
226468  Exiting thread 8 after 0.18s.
233649  Exiting thread 3 after 0.23s.
234103  Exiting thread 0 after 0.23s.
238323  Exiting thread 9 after 0.18s.
```

Concurrency

Example: threadHello

```
#import <pthread.h>

#define NTHREADS 10

thread_t threads[NTHREADS];

int main() {
  for (i = 0; i < NTHREADS; i++)
    thread_create(&threads[i], &go, i);
  for (i = 0; i < NTHREADS; i++) {
    exitValue = thread_join(threads[i]);
    printf("Thread %d returned with %ld\n",
          i, exitValue);
  }
  printf("Main thread done.\n");
}

void go (int n) {
  printf("Hello from thread %d\n", n);
  thread_exit(100 + n);
  // REACHED?
}
```

threadHello: Example Output

- Why must "thread returned" print in order?
- What is maximum # of threads running when thread 5 prints hello?
- Minimum?

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

Fork/Join Concurrency

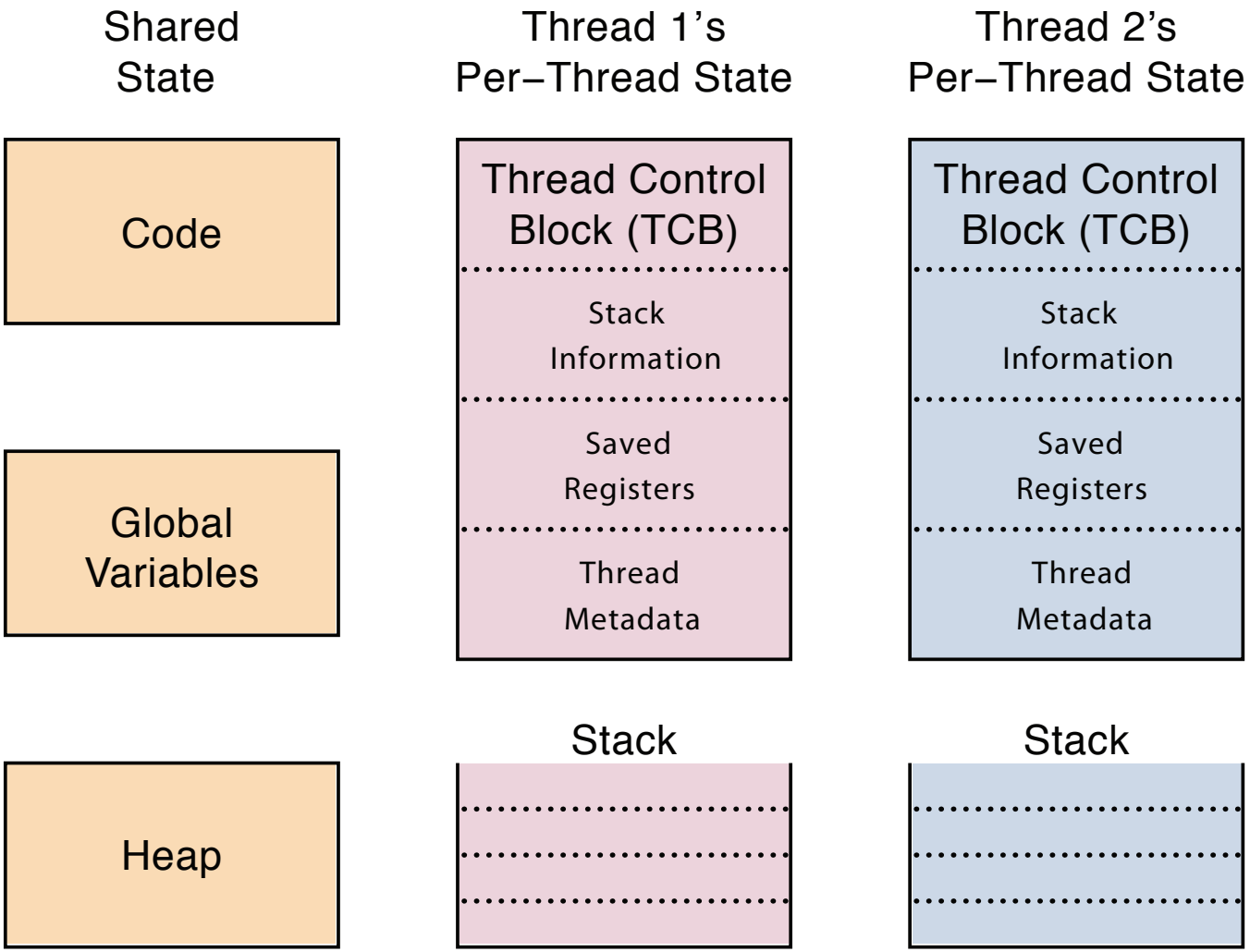
- Threads can create children, and wait for their completion
- Data only shared before fork/after join
- Examples:
  - Web server: fork a new thread for every new connection
    - As long as the threads are completely independent
  - Merge sort
  - Parallel memory copy

bzero with fork/join concurrency

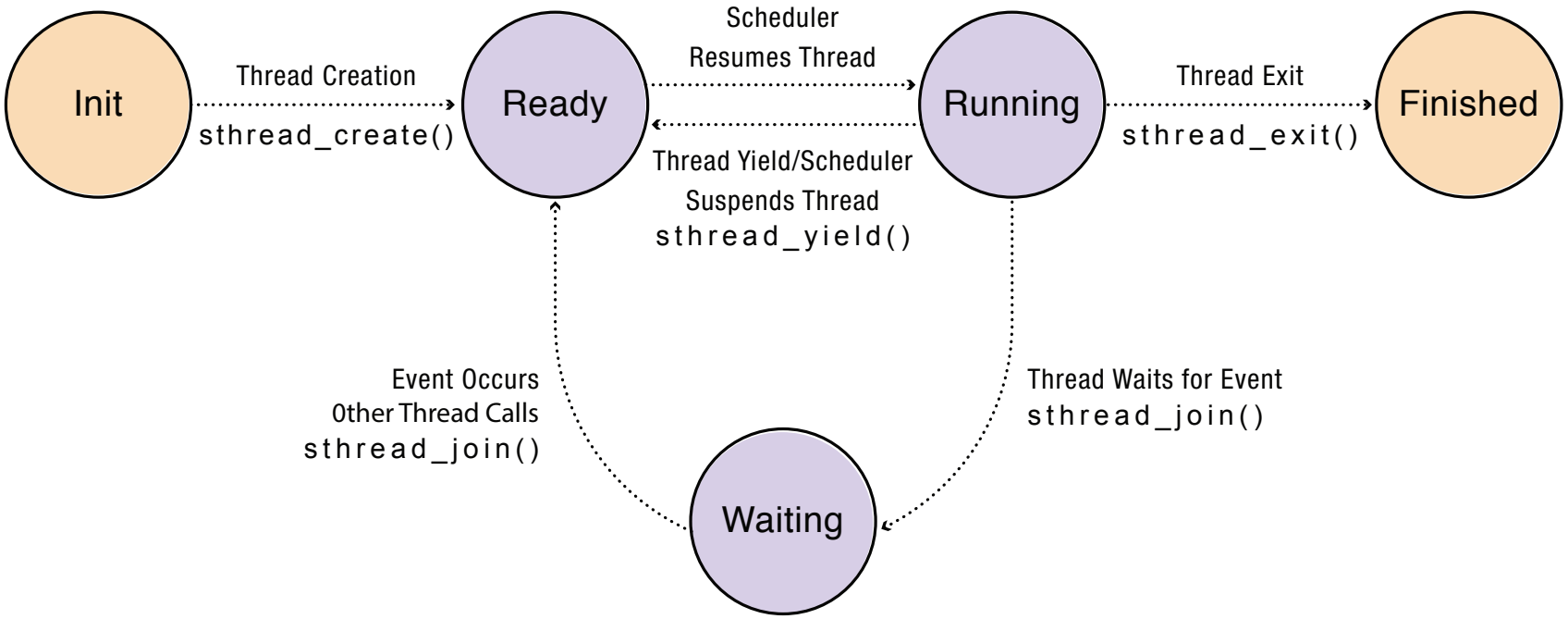
```
void blockzero (unsigned char *p, int length) {
  int i, j;
  thread_t threads[NTHREADS];
  struct bzeroparams params[NTHREADS];

  // For simplicity, assumes length is divisible by NTHREADS.
  for (i = 0, j = 0; i < NTHREADS; i++, j += length/NTHREADS) {
    params[i].buffer = p + i * length/NTHREADS;
    params[i].length = length/NTHREADS;
    thread_create_p(&(threads[i]), &go, &params[i]);
  }
  for (i = 0; i < NTHREADS; i++) {
    thread_join(threads[i]);
  }
}
```

Thread Data Structures



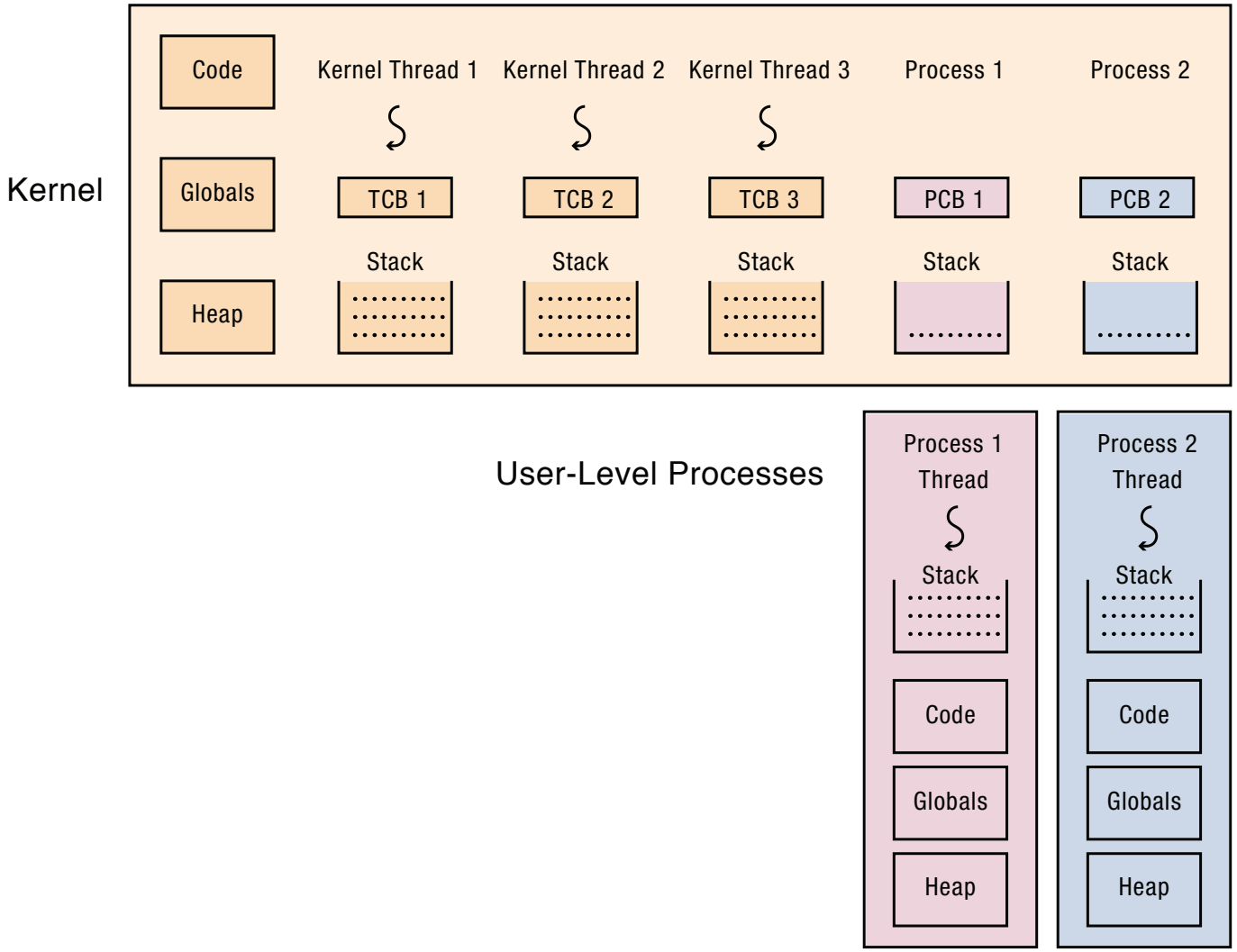
Thread Lifecycle



Implementing Threads: Roadmap

- Kernel threads
  - Thread abstraction only available to kernel
  - To the kernel, a kernel thread and a single threaded user process look quite similar
- Multithreaded processes using kernel threads (Linux, MacOS)
  - Kernel thread operations available via syscall
- User-level threads
  - Thread operations without system calls

Multithreaded OS Kernel



Implementing threads

- Fork implementation
  - Allocate thread control block
  - Allocate stack
  - Build stack frame for base of stack (fake stub call)
  - Put function , args on stack
  - Put thread on ready list
- Stub function implementation (why is it needed?)
  - Call the actual function
  - If return, call thread exit function

Thread Context Switch

- Voluntary
  - During normal operation
    - Yeld control to another thread in the ready list
  - During join
    - Yeld control to another thread in the ready list, if child is not done yet
- Involuntary
  - Interrupt or exception
  - Some other thread is higher priority

Voluntary thread context switch

- Implementation
  - Save registers on old stack
  - Switch to new stack, new thread
  - Restore registers from new stack
  - Return
- Exactly the same with kernel threads or user threads

Involuntary Thread/Process Switch

- Timer or I/O interrupt
  - Tells OS some other thread should run
- Simple version
  - End of interrupt handler calls thread switch
  - When resumed, return from handler resumes kernel thread or user process
  - Thus, processor context is saved/restored twice (once by interrupt handler, once by thread switch)

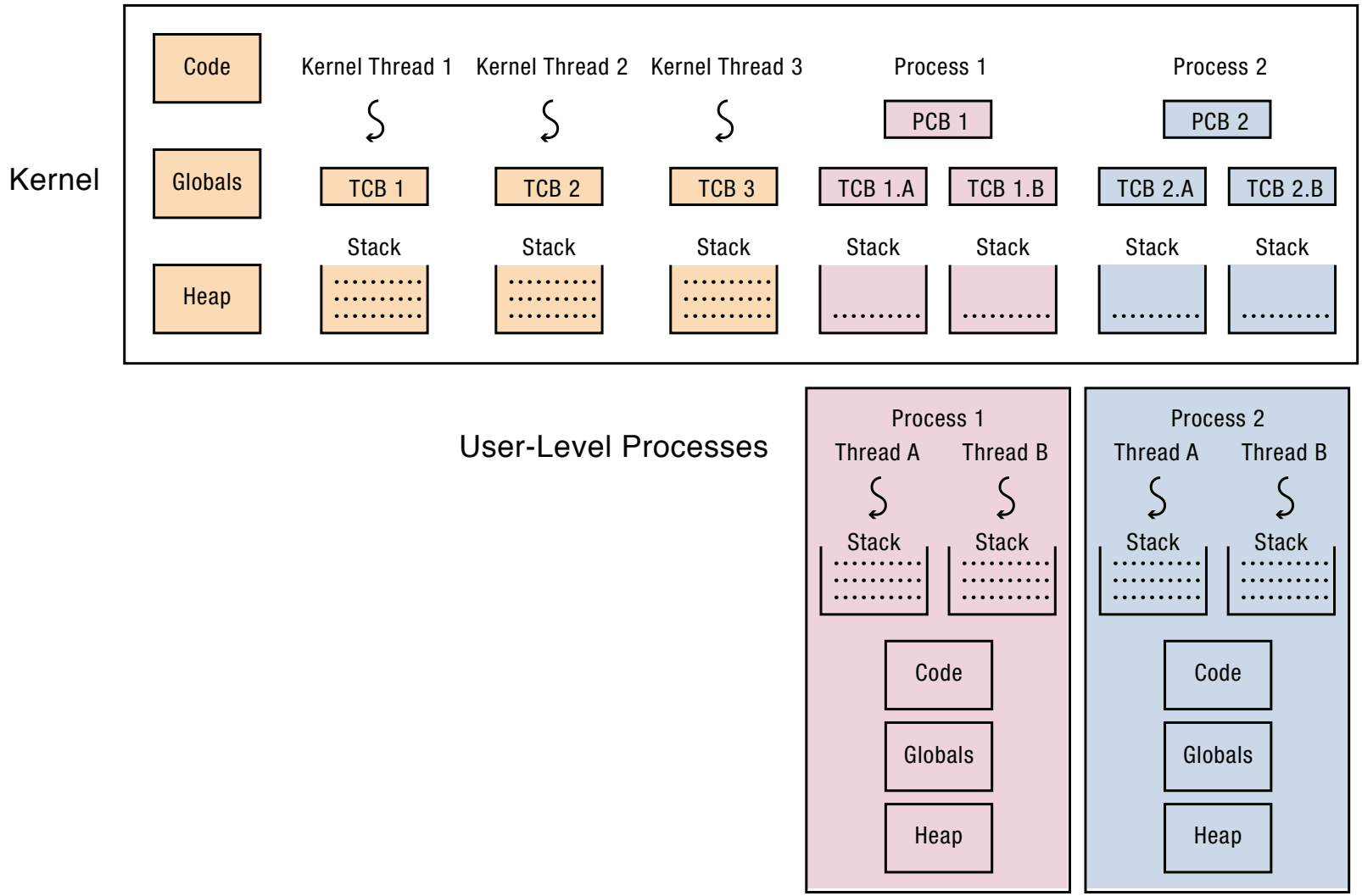
Faster Thread/Process Switch

- What happens on a timer (or other) interrupt?
  - Interrupt handler saves state of interrupted thread
  - Decides to run a new thread
  - Throw away current state of interrupt handler!
  - Instead, set saved stack pointer to trapframe
  - Restore state of new thread
  - On resume, pops trapframe to restore interrupted thread

Multithreaded User Processes (Take 1)

- User thread = kernel thread
  - System calls for thread fork, join, exit (and lock, unlock,. . .)
  - Kernel does context switch
  - Simple, but a lot of transitions between user and kernel mode

Multithreaded User Processes (Take 1)



Multithreaded User Processes (Take 2)

- Green threads (early Java)
  - User-level library, within a single-threaded process
  - Library does thread context switch
  - Preemption via upcall/UNIX signal on timer interrupt
  - Use multiple processes for parallelism
    - Shared memory region mapped into each process

Multithreaded User Processes (Take 3)

- Scheduler activations
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - Thread library decides what thread to run next
  - Upcall whenever kernel needs a user-level scheduling decision
    - Process assigned a new processor
    - Processor removed from process
    - System call blocks in kernel

Multicore and multithreading

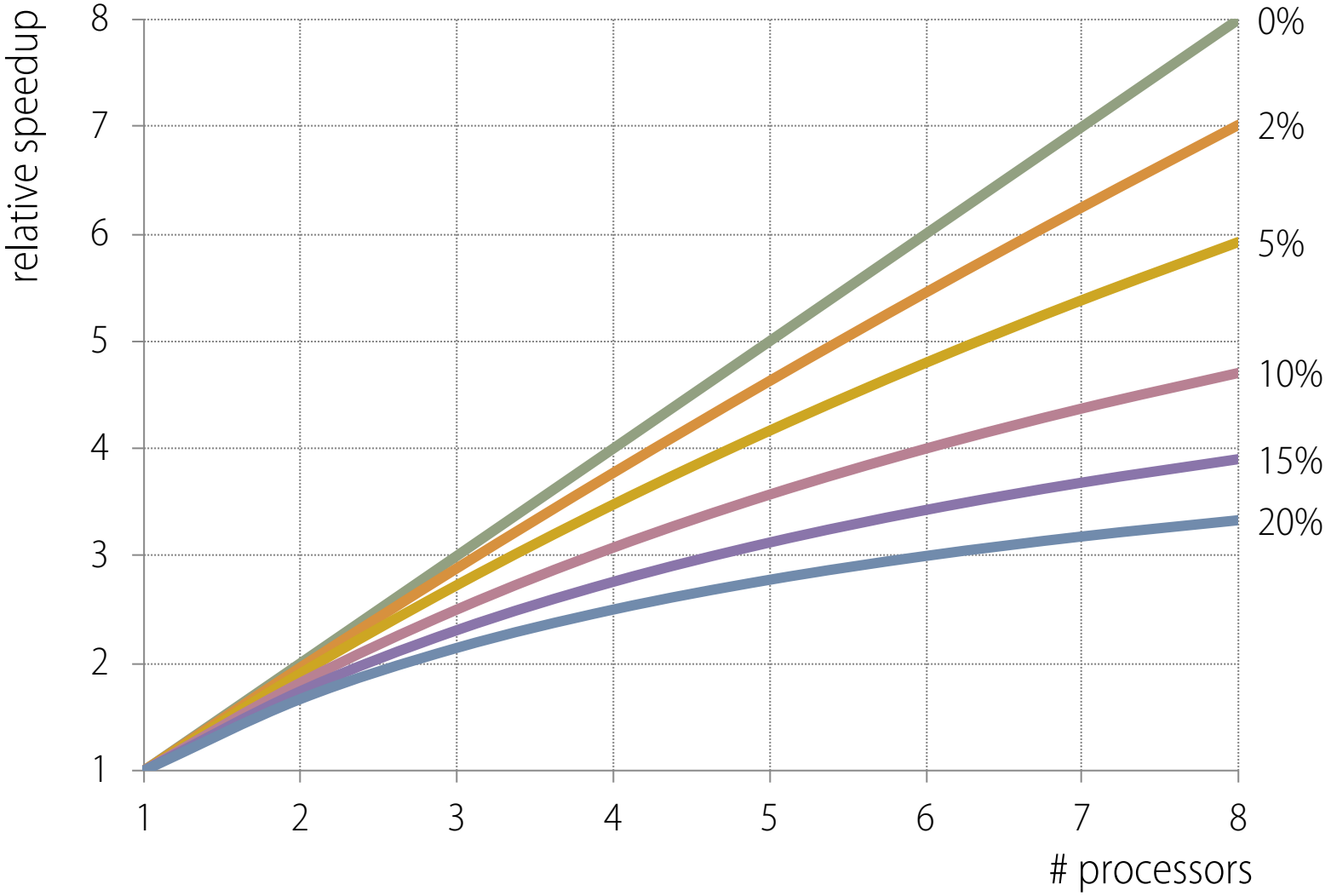
Amdahl's Law (1967) and the ability to scale

- Deals with the potential speedup of a program using multiple processors compared to a single processor.
- Assume that, of a program's total execution time in a single processor,
  - a fraction f involves code that is infinitely parallelizable with no overhead
  - a fraction $(1-f)$ involves code that is inherently serial
- Then the speedup using a parallel processor that fully exploits the parallel portion of the program is as follows:

$$\text{Speedup} = \frac{\text{time to execute on a single processor}}{\text{time to execute on } N \text{ parallel processors}} = \frac{1}{(1-f) + \frac{f}{N}}$$

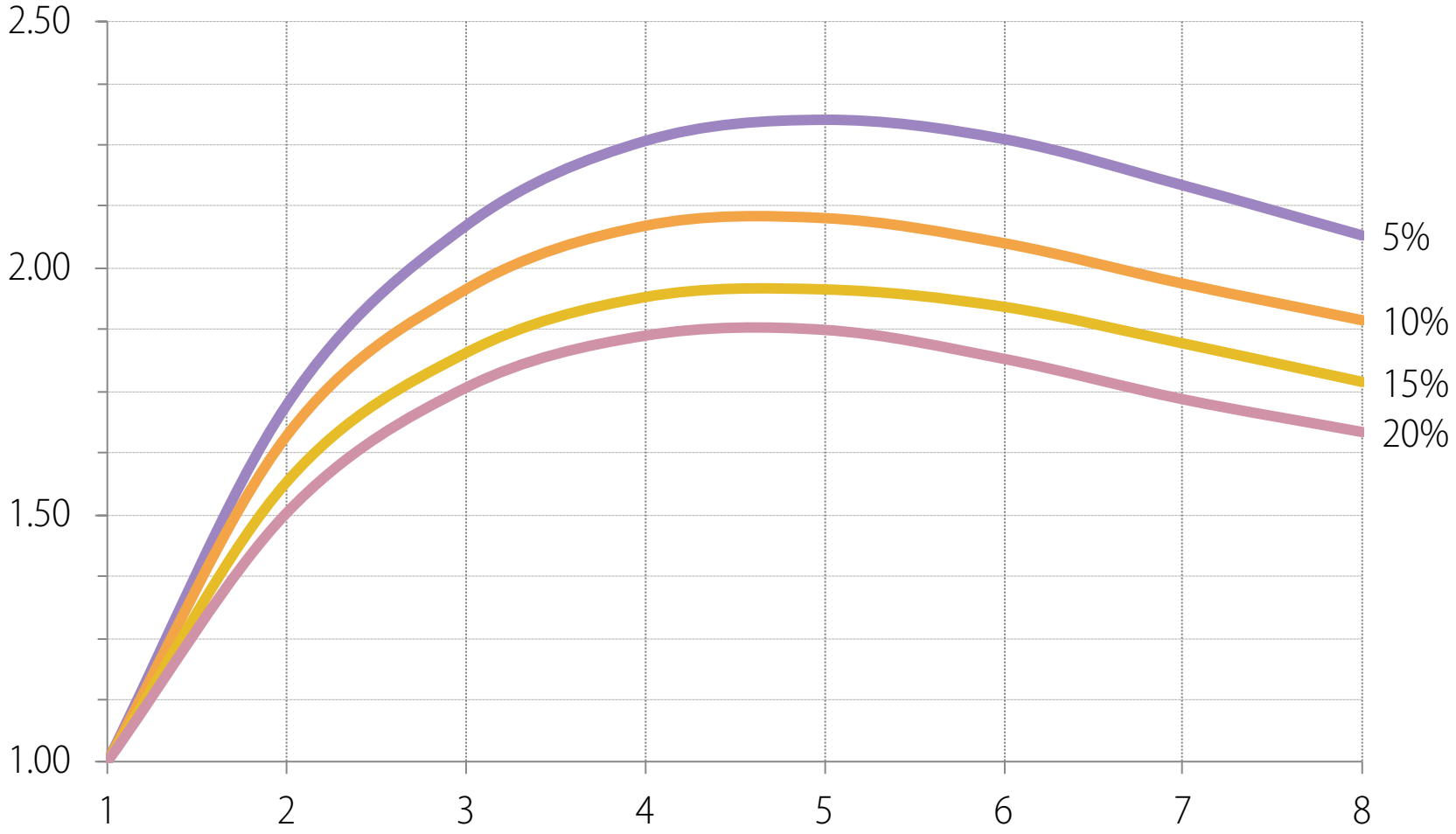
Performance effect of multiple cores

(speedup with x% sequential portion, without overheads)

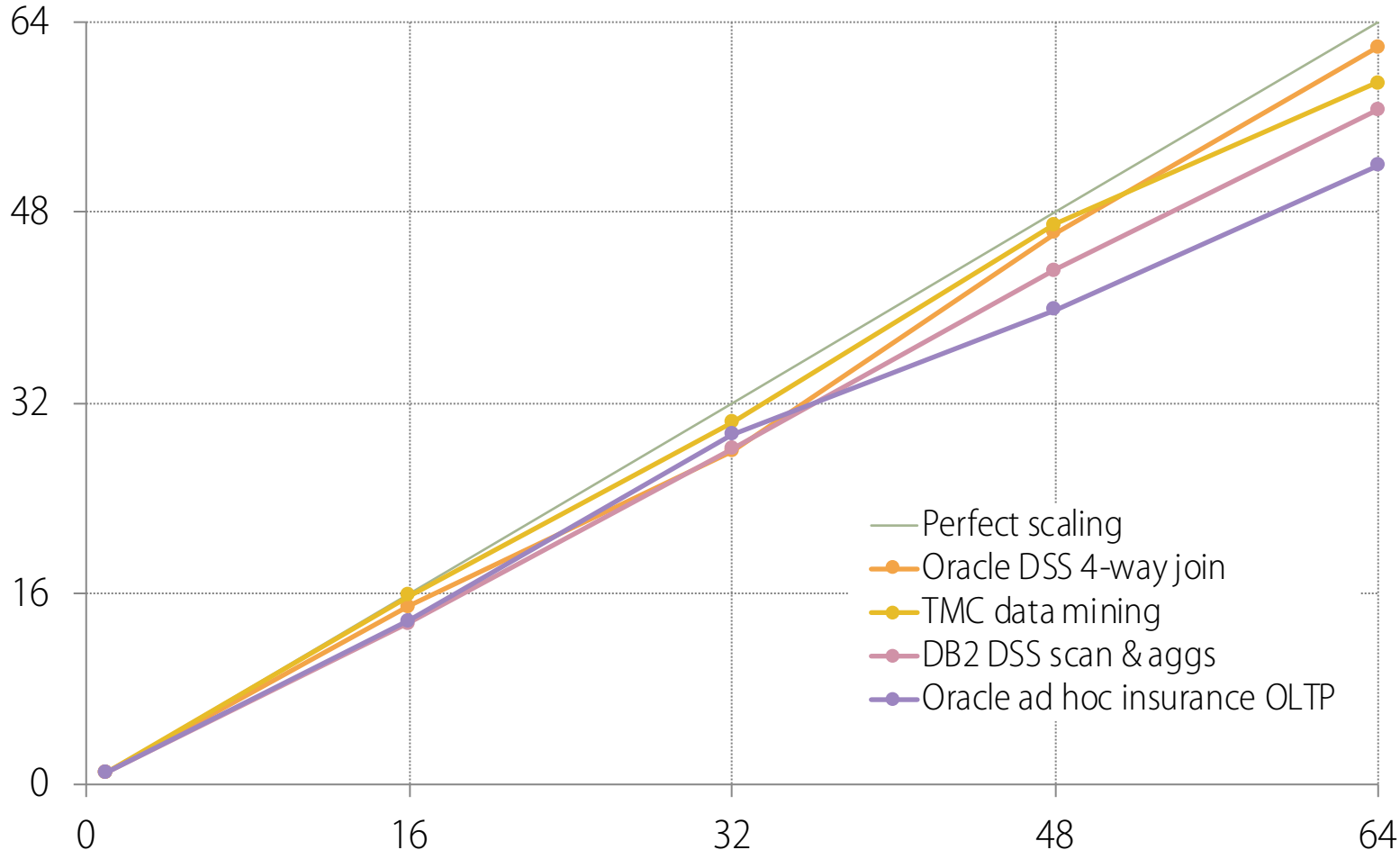


Performance effect of multiple cores

(speedup with x% sequential portion, including overheads)



Scaling of database workloads on multiprocessor hardware



Scaling throughput with the number of cores

- Classes of applications that benefit directly from such ability
  - General purpose server software
  - Multithreaded native applications
  - Multiprocess applications
  - Java applications
  - Multiinstance applications

Threading issues

Threading Issues

- Semantics of fork() system call
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations

Threading Issues

- Semantics of fork() system call
  - Does fork() duplicate only the calling thread or all threads?
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations

Threading Issues

- Semantics of fork() system call
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations

- There are two general approaches:
  - Asynchronous cancellation terminates the target thread immediately.
  - Deferred cancellation allows the target thread to periodically check if it should be cancelled.
    - This provides for orderly thread cancellation.
    - Checkpoints are called cancellation points.

Threading Issues

- Semantics of fork() system call
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations

- A signal is issued to notify the occurrence of a particular event
- A handler delivers a given signal to
  - The thread to which it applies
  - Every thread in the process
  - Certain threads in the process
  - A specific thread that is assigned to receive all signals for the process

Threading Issues

- Semantics of fork() system call
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations

- To avoid unlimited threads which could exhaust system resources, a server creates a number of threads in a pool where they await work.
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread.
  - The number of threads in the applications can be bound to the size of the pool.

Threading Issues

- Semantics of fork() system call
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations

- A thread may be allowed to have its own copy of certain data
- Useful when you cannot control the thread creation process (e.g., when thread pools are used)

Threading Issues

- Semantics of fork() system call
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations

- The "combined" model requires communication between the kernel and the thread library to maintain the appropriate number of kernel threads allocated to the application.
- In a *scheduler activation* scheme the kernel provides an application with a set of virtual processors on which the application can schedule user threads.
- Scheduler activations also provide *upcalls* – a communication mechanism from the kernel to the thread library to inform the application about certain events.