CS 423
Operating System Design:
Concurrency

Tianyin Xu

* Thanks for Prof. Adam Bates for the slides.

Nix and NixOS

- What is nix
  - Nix is a powerful *package manager* for Linux and other Unix systems that makes package management reliable and reproducible. It provides atomic upgrades and rollbacks, side-by-side installation of multiple versions of a package, multi-user package management and easy setup of build environments.
- What is nixOS?
  - NixOS is a Linux *distribution* with a unique approach to package and configuration management. Built on top of the Nix package manager, it is completely declarative, makes upgrading systems reliable, and has many other advantages.

NixOS

- What's special about Nix (not yet another boring package manager like Apt and Yum)?
  - Declarative
  - Reliable upgrade
    - Atomic
    - Rollback
    - Reproducible
  - Safe to test changes
  - (some more; but not that impressive)

It was a research project.

Nix: A Safe and Policy-Free System for Software Deployment

Eelco Dolstra, Merijn de Jonge, and Eelco Visser – Utrecht University

ABSTRACT

Existing systems for software deployment are neither safe nor sufficiently flexible. Primary safety issues are the inability to enforce reliable specification of component dependencies, and the lack of support for multiple versions or variants of a component. This renders deployment operations such as upgrading or deleting components dangerous and unpredictable. A deployment system must also be flexible (i.e., policy-free) enough to support both centralised and local package management, and to allow a variety of mechanisms for transferring components. In this paper we present Nix, a deployment system that addresses these issues through a simple technique of using cryptographic hashes to compute unique paths for component instances.

Introduction

Software deployment is the act of transferring software to the environment where it is to be used. This is a deceivingly hard problem: a number of requirements make effective software deployment difficult in practice, as most current systems fail to be sufficiently *safe* and *flexible*.

The main safety issue that a software deployment system must address is *consistency*: no deployment action should bring the set of installed software components into an inconsistent state. For instance, an installed component should never be able to refer to any component not present in the system; and upgrading or removing components should not break other components or running programs [15], e.g., by overwriting the files of those components. In particular, it should be possible to have multiple versions and variants of a component installed at the same time. No duplicate components should be installed: if two components have a shared dependency, that dependency should be stored exactly once.

Deployment systems must be flexible. They should support both *centralised* and *local package management*: it should be possible for both site administrators and local users to install applications, for instance, to be able to use different versions and variants of components. Finally, it must not be difficult to support deployment both in source and binary form, or to define a variety of mechanisms for transferring components. In other words, a deployment system should provide flexible *mechanisms*, not rigid *policies*.

Despite much research in this area, proper solutions have not yet been found. For instance, a summary of twelve years of research in this field indicates, amongst others, that many existing tools ignore the problem of interference between components and that end-user customisation has only been slightly examined [6]. Consequently, there are still many hard outstanding deployment problems (see the first section), and there seems to be no general deployment system available that satisfies all the above requirements. Most existing tools only consider a small subset of these requirements and ignore the others.

In this paper we present Nix, a safe and flexible deployment system providing mechanisms that can be used to define a great variety of deployment policies. The primary features of Nix are:

- Concurrent installation of multiple versions and variants
- Atomic upgrades and downgrades
- Multiple user environments
- Safe dependencies
- Complete deployment
- Transparent binary deployment as an optimisation of source deployment
- Safe garbage collection
- Multi-level package management (i.e., different levels of centralised and local package management)
- Portability

These features follow from the fairly simple technique of using cryptographic hashes to compute unique paths for component instances.

Motivation

In this section we take a close look at the issues that a system for software deployment must be able to deal with.

Dependencies For safe software deployment, it is essential that the *dependencies* of a component are correctly identified. For correct deployment of a component, it is necessary not only to install the component itself, but also all components which it may need. If the identification of dependencies is incomplete, then the component may or may not work, depending on whether the omitted dependencies are already present on the target system. In this case, deployment is said to be *incomplete*.

Integrating Software Construction and Software Deployment

Eelco Dolstra

Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
eelco@cs.uu.nl

Abstract. Classically, software deployment is a process consisting of building the software, packaging it for distribution, and installing it at the target site. This approach has two problems. First, a package must be annotated with dependency information and other meta-data. This to some extent overlaps with component dependencies used in the build process. Second, the same source system can often be built into an often very large number of *variants*. The distributor must decide which element(s) of the variant space will be packaged, reducing the flexibility for the receiver of the package. In this paper we show how building and deployment can be integrated into a single formalism. We describe a build manager called *Maak* that can handle deployment through a sufficiently general module system. Through the sharing of generated files, a source distribution *transparently* turns into a binary distribution, removing the dichotomy between these two modes of deployment. In addition, the creation and deployment of variants becomes easy through the use of a simple functional language as the build formalism.

1 Introduction

Current SCM systems treat the building of software and the deployment of software as separate, orthogonal steps in the software life-cycle. Controlling the former is the domain of tools such as Make [1], while the latter is handled by, e.g., the Red Hat Package Manager [2]. In fact, they are not orthogonal. In this paper we show how building and deployment can be integrated in an elegant way. There are a number of problems in the current approaches to building and deployment.

Component dependencies Separating the building and deployment steps leads to a discontinuity in the formalisms used to express component dependencies. In a build manager it is necessary to express the dependencies between source components; in a package manager we express the dependencies between binary components.

Source vs. Binary Distribution Another issue is the dichotomy between source and binary distributions. In an open-source environment software is provided as

Research -> Impact

- Xen (Intel/Linux Foundation)
- Google (Google)
- Spark (Databricks)
- PatternInsight (acquired by VMWare)
- Veriflow (acquired by VMWare)

Name

The name Nix is derived from the Dutch word niks, meaning nothing; build actions do not see anything that has not been explicitly declared as an input.

Remind me of:

P: *probeer te verlagen (try to decrease)*

V: *verhogen (increase)*

Concurrency vs Parallelism

Two tasks

1. Get a visa
2. Prepare slides

1. Sequential execution
2. Concurrent execution
3. Parallel execution
4. Concurrent but not parallel
5. Parallel but not concurrent
6. Parallel and concurrent

Why Concurrency?

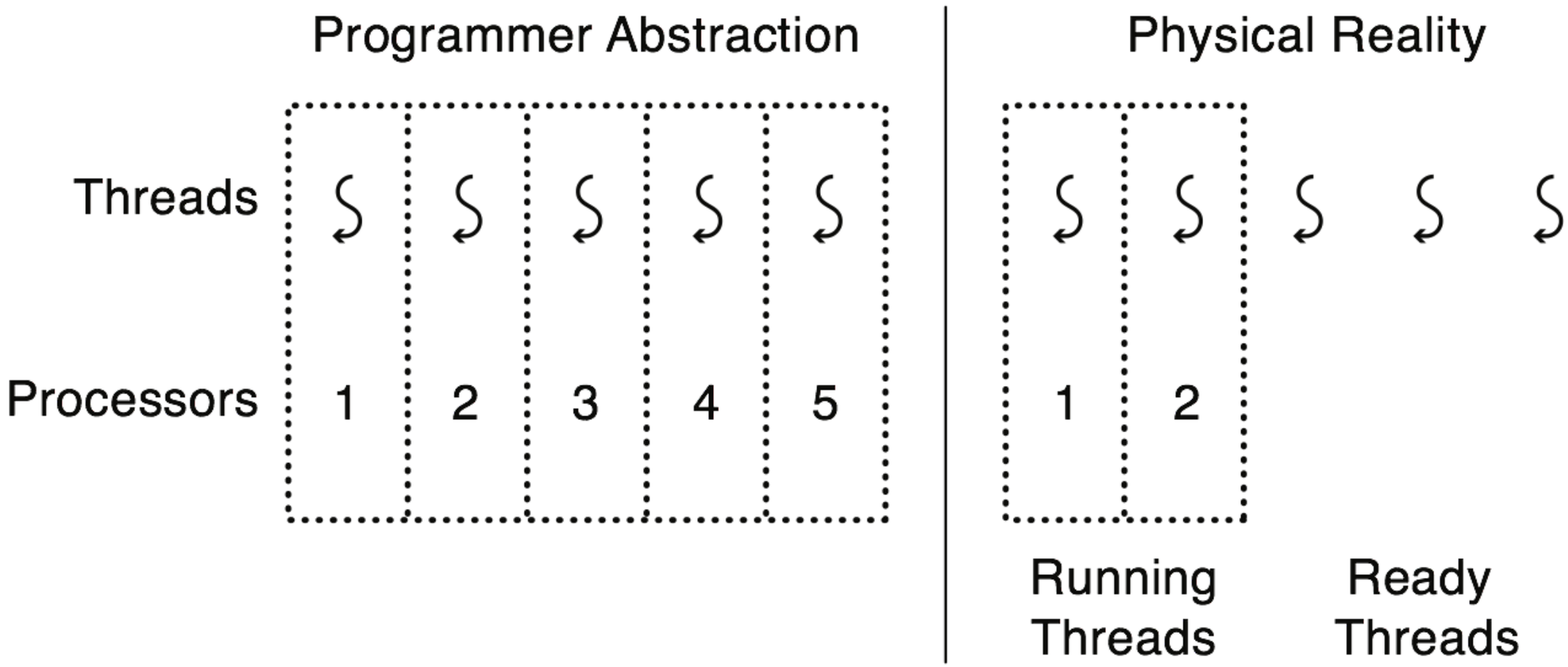
- Servers
  - Multiple connections handled simultaneously
- Parallel programs
  - To achieve better performance
- Programs with user interfaces
  - To achieve user responsiveness while doing computation
- Network and disk bound programs
  - To hide network/disk latency

Definitions

- Thread: A single execution sequence that represents a separately schedulable task.
  - *Single execution sequence*: intuitive and familiar programming model
  - *separately schedulable*: OS can run or suspend a thread at any time.
  - Schedulers operate over threads/tasks, both kernel and user threads.
- *Does the OS protect all threads from one another?*

The Thread Abstraction

- Infinite number of processors
- Threads execute with variable speed



Programmer vs. Processor View

Programmer View

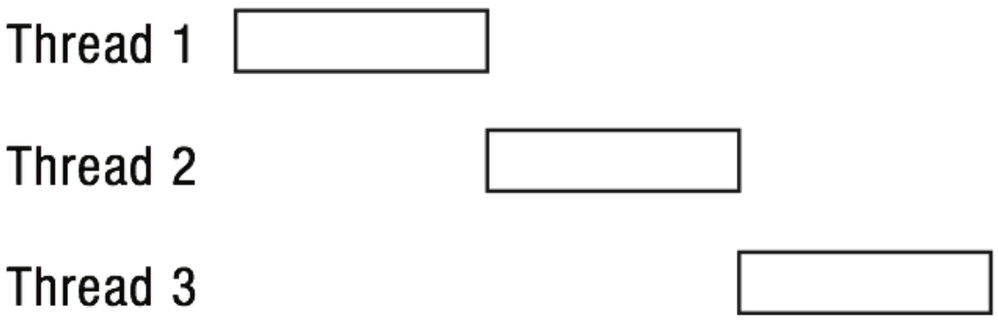
Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	.....	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended.	.....
.	.	Other thread(s) run.	Thread is suspended.
.	.	Thread is resumed.	Other thread(s) run.
.	.	.....	Thread is resumed.
		y = y + x;	.....
		z = x + 5y;	z = x + 5y;

Variable Speed: Program must anticipate all of these possible executions

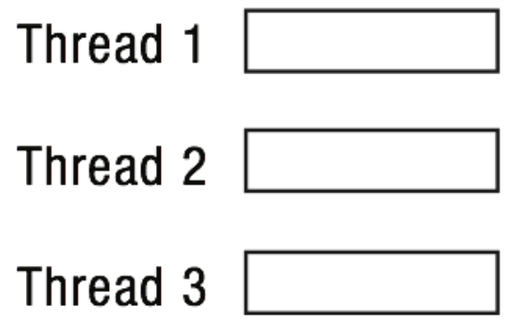
Possible Executions

Processor View

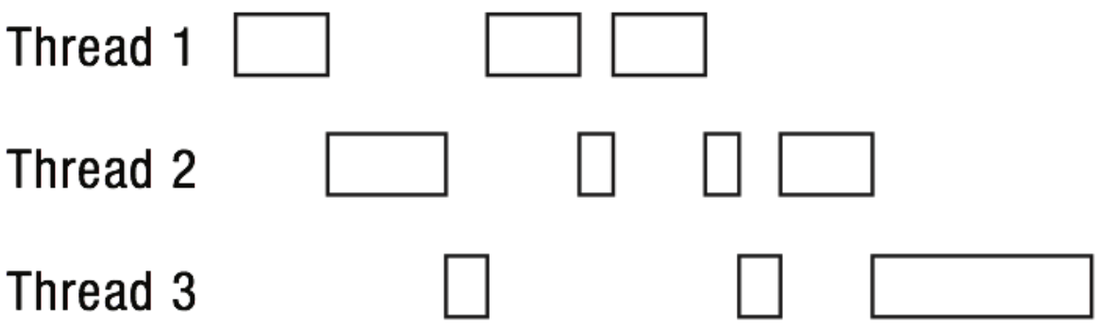
One Execution



Another Execution



Another Execution



Something to look forward to when we discuss scheduling!

Thread Ops

- thread_create(thread, func, args)
  Create a new thread to run func(args)

- thread_yield()
  Relinquish processor voluntarily

- thread_join(thread)
  In parent, wait for forked thread to exit, then return

- thread_exit
  Quit thread and clean up, wake up joiner if any

Ex: threadHello

```
#define NTHREADS 10
thread_t threads[NTHREADS];
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?
}
```

Ex: threadHello output

```
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.
```

- Must "thread returned" print in order?
- What is maximum # of threads that exist when thread 5 prints hello?
- Minimum?
- Why aren't any messages interrupted mid-string?

Create/Join Concurrency

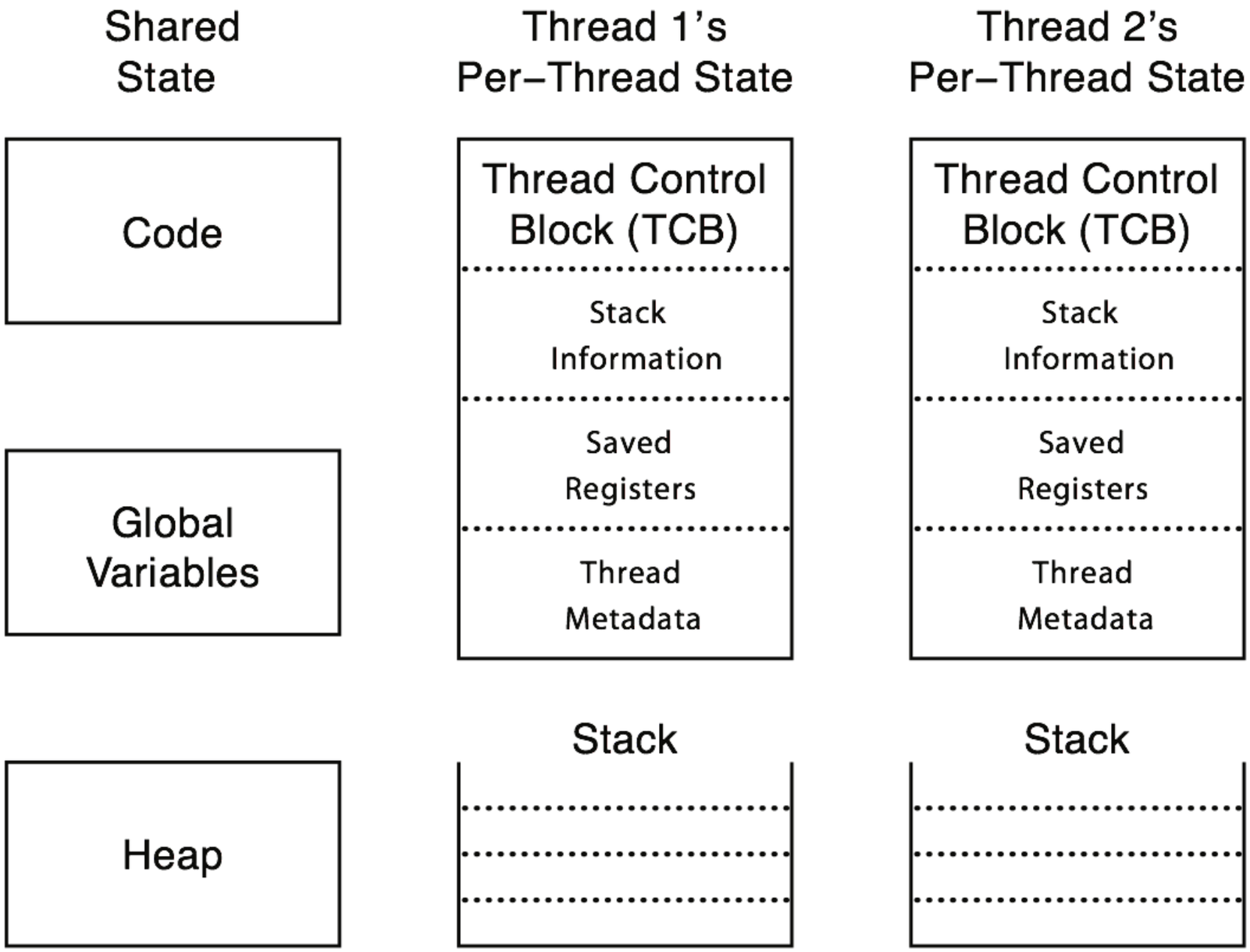
- Threads can create children, and wait for their completion
- Data only shared before fork/after join
- Examples:
  - Web server: fork a new thread for every new connection
    - As long as the threads are completely independent
  - Merge sort
  - Parallel memory copy

Ex: bzero

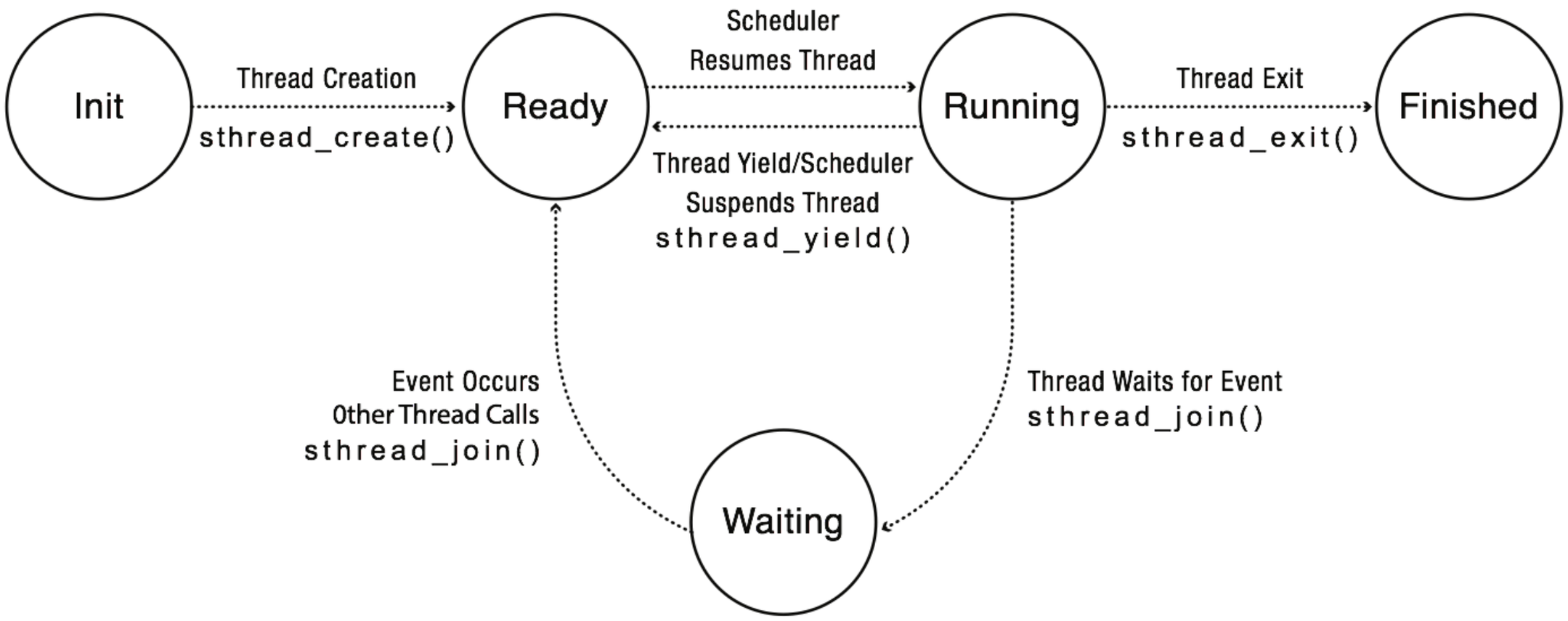
```
void blockzero (unsigned char *p, int length) {
    int i, j;
    thread_t threads[NTHREADS];
    struct bzeroparams params[NTHREADS];

// 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



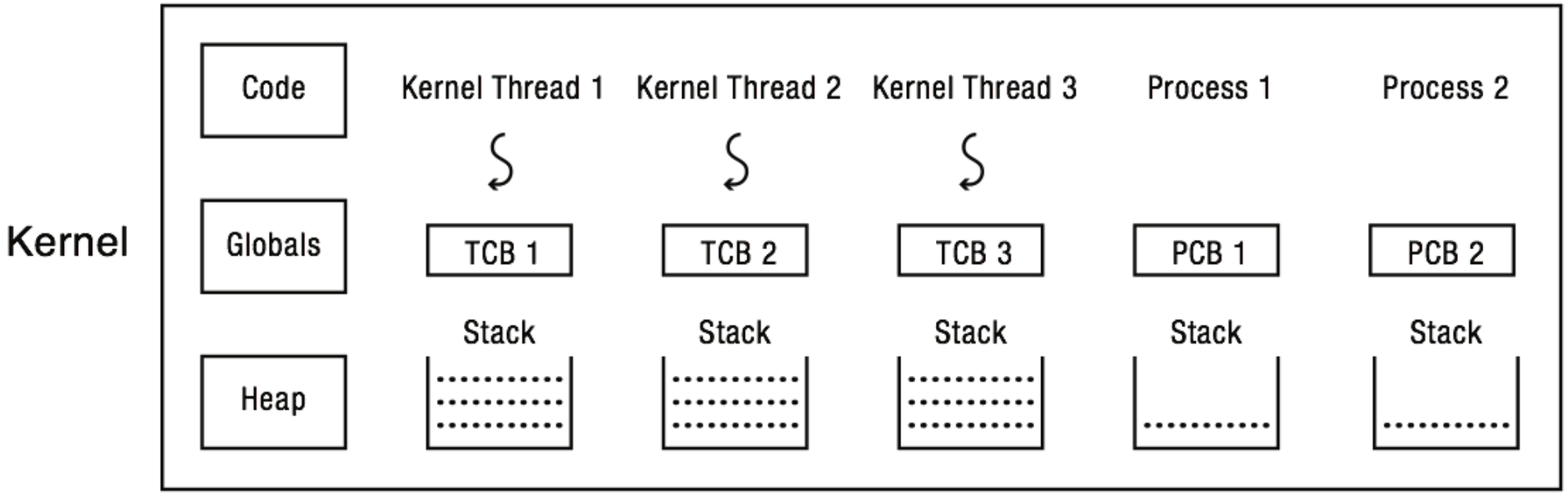
Thread Lifecycle

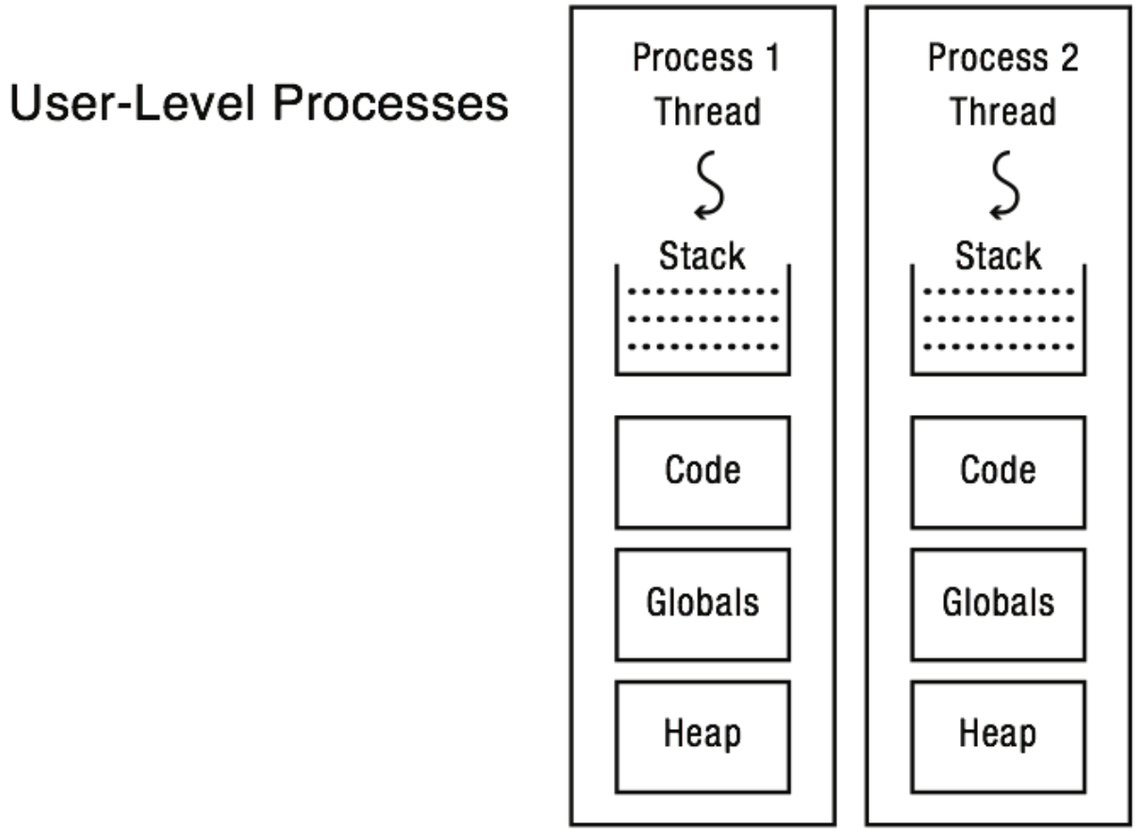


Thread Implementations

- 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
  - Kernel thread operations available via syscall
- User-level threads
  - Thread operations without system calls

Multithreaded OS Kernel





Implementing Threads

- Thread_fork(func, args)
  - Allocate thread control block
  - Allocate stack
  - Build stack frame for base of stack (stub)
  - Put func, args on stack
  - Put thread on ready list
  - Will run sometime later (maybe right away!)

- stub(func, args):
  - Call (*func)(args)
  - If return, call thread_exit()

Implementing Threads

- Thread_Exit
  - Remove thread from the ready list so that it will never run again
  - Free the per-thread state allocated for the thread

switchframe

How do we switch out thread state? (i.e., ctx switch)

```
# Save caller's register state
#  NOTE: %eax, etc. are ephemeral
pushl %ebx
pushl %ebp
pushl %esi
pushl %edi

# Get offsetof (struct thread, stack)
mov thread_stack_ofs, %edx
# Save current stack pointer to old thread's stack, if any.
movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax,%edx,1)

# Change stack pointer to new thread's stack
# this also changes currentThread
movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp

# Restore caller's register state.
popl %edi
popl %esi
popl %ebp
popl %ebx
ret
```

Ex: Two Threads call Yield

Thread 1's instructions	**Thread 2's instructions**	**Processor's instructions**
"return" from thread_switch into stub		"return" from thread_switch into stub
call go		call go
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch
save thread 1 state to TCB		save thread 1 state to TCB
load thread 2 state		load thread 2 state
	"return" from thread_switch into stub	"return" from thread_switch into stub
	call go	call go
	call thread_yield	call thread_yield
	choose another thread	choose another thread
	call thread_switch	call thread_switch
	save thread 2 state to TCB	save thread 2 state to TCB
	load thread 1 state	load thread 1 state
return from thread_switch		return from thread_switch
return from thread_yield		return from thread_yield
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch

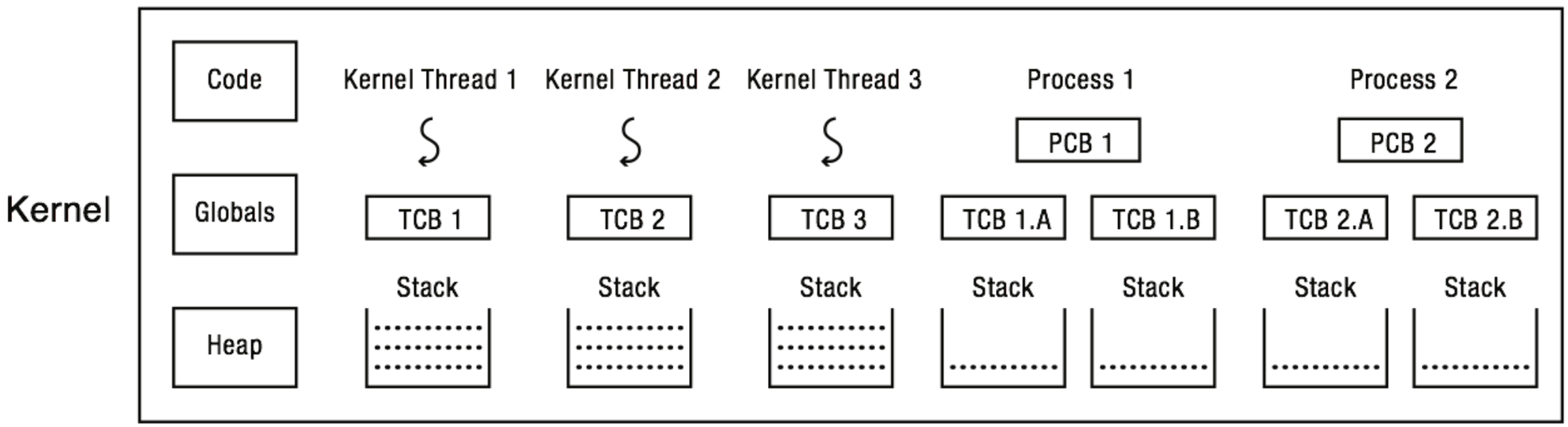
Multi-threaded User Processes

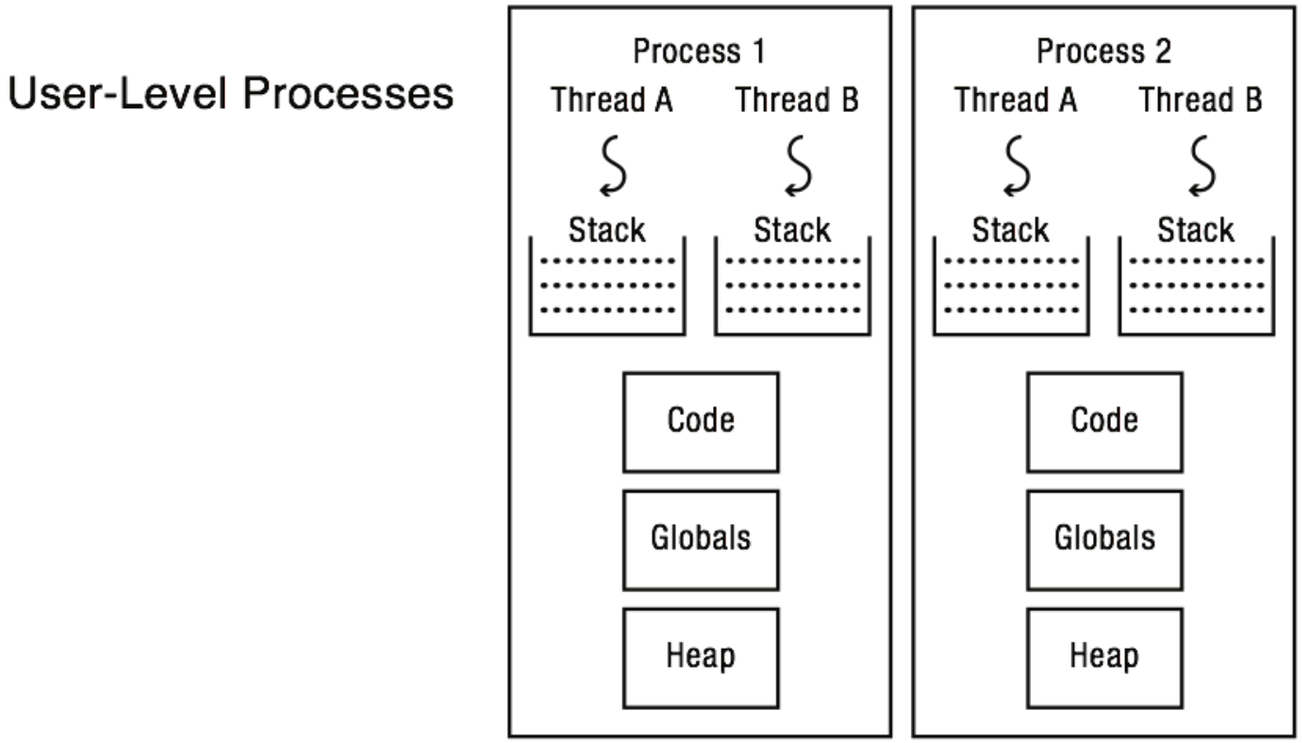
Take 1:

- User thread = kernel thread (Linux, MacOS)
  - 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

Multi-threaded User Processes

Take 1:





Multi-threaded 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

Multi-threaded User Processes

Take 3:

- Scheduler activations (Windows 8):
  - 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

Multi-threaded User Processes

Take 3: (What's old is new again)

M:N model multiplexes N user-level threads onto M kernel-level threads

Good idea? Bad Idea?

Question

Compare event-driven programming with multithreaded concurrency. Which is better in which circumstances, and why?