CS 240

Simple Introduction to Threads (I)

(pthreads)

What is a (unix) process ?

- Every a.out becomes a process (instance of a running program)
- All Unix commands (cc, mail) run as processes
- Each processes gets a CPU time-slice to run
- Despite this, user sees “smooth”execution
- CPU sees two separate threads of execution when proc A and proc B interleave

Summary: going from a.out to a process

How with a picture

What does a process need? How is it laid out in memory?

A process needs:

- a stack
- instructions to execute (text)
- data
- heap
- registers
- resources (files, signals, sockets, locks)

LAYOUT

How is a thread different from a process?

When a.outx and a.outy run, CPU sees two threads of execution
Each process has its own address space; no shared variables

A thread is an ADT representing flow of control within a process.
Loosely, it is used to refer to two things:

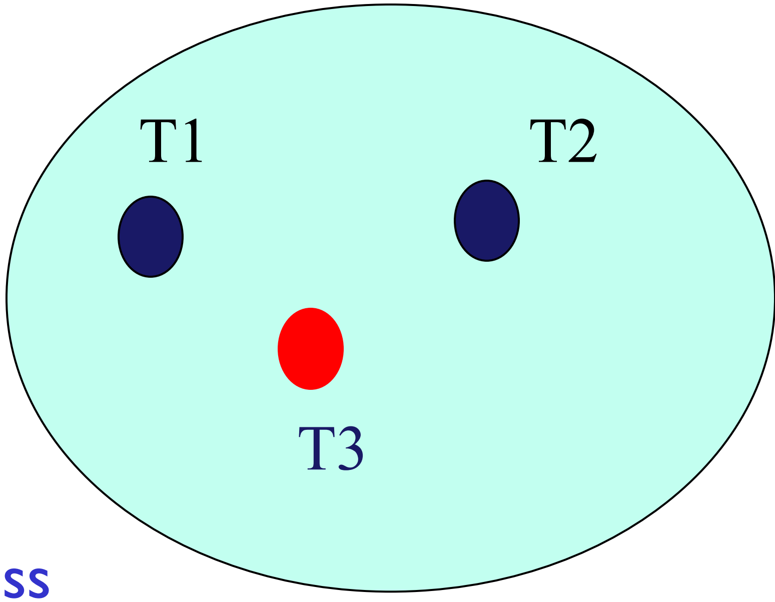
1. one is an abstraction (idea that different computations interleave), and
2. the other is a construct (an actual thread entity, as in pthreads)

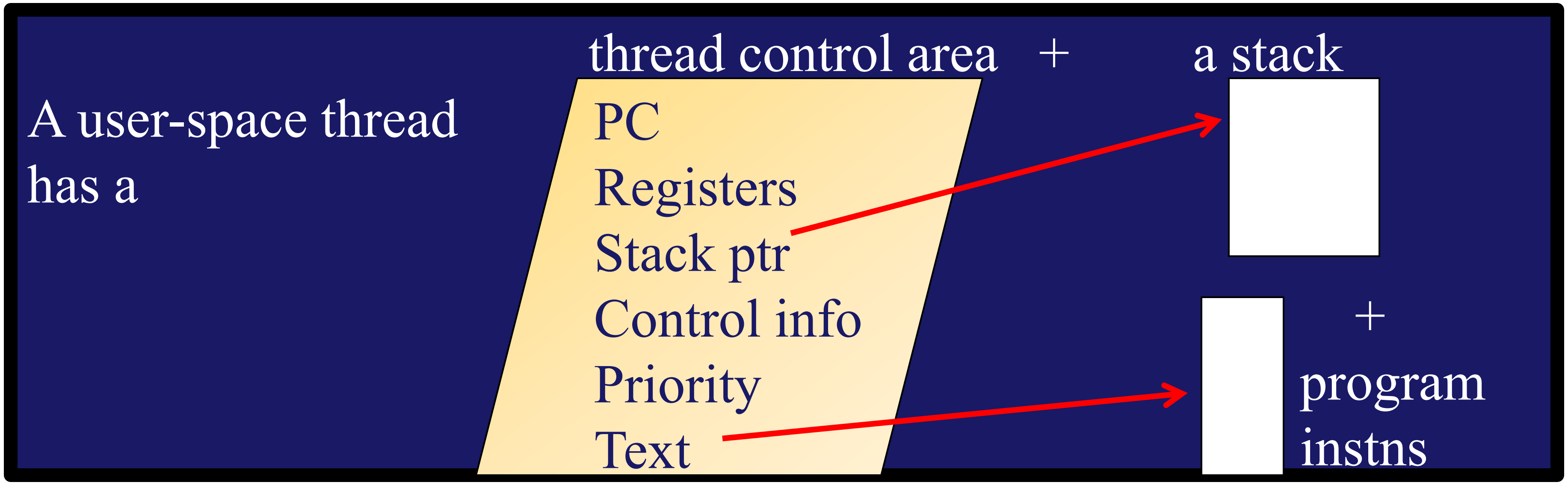
Before we had threads libraries, we would use processes to get distinct threads of execution. How? Many processes, each with its own state and ID for management and control.

A thread executes within a process; so a process acts to group resources together while a thread is something the CPU schedules.

User-space Threads

- start a process: a.out
- create some threads
- only user-libs, no syscalls
- switching between threads
  does not involve OS interrupts
- kernel only sees process, not threads
- you manage the threads inside the process

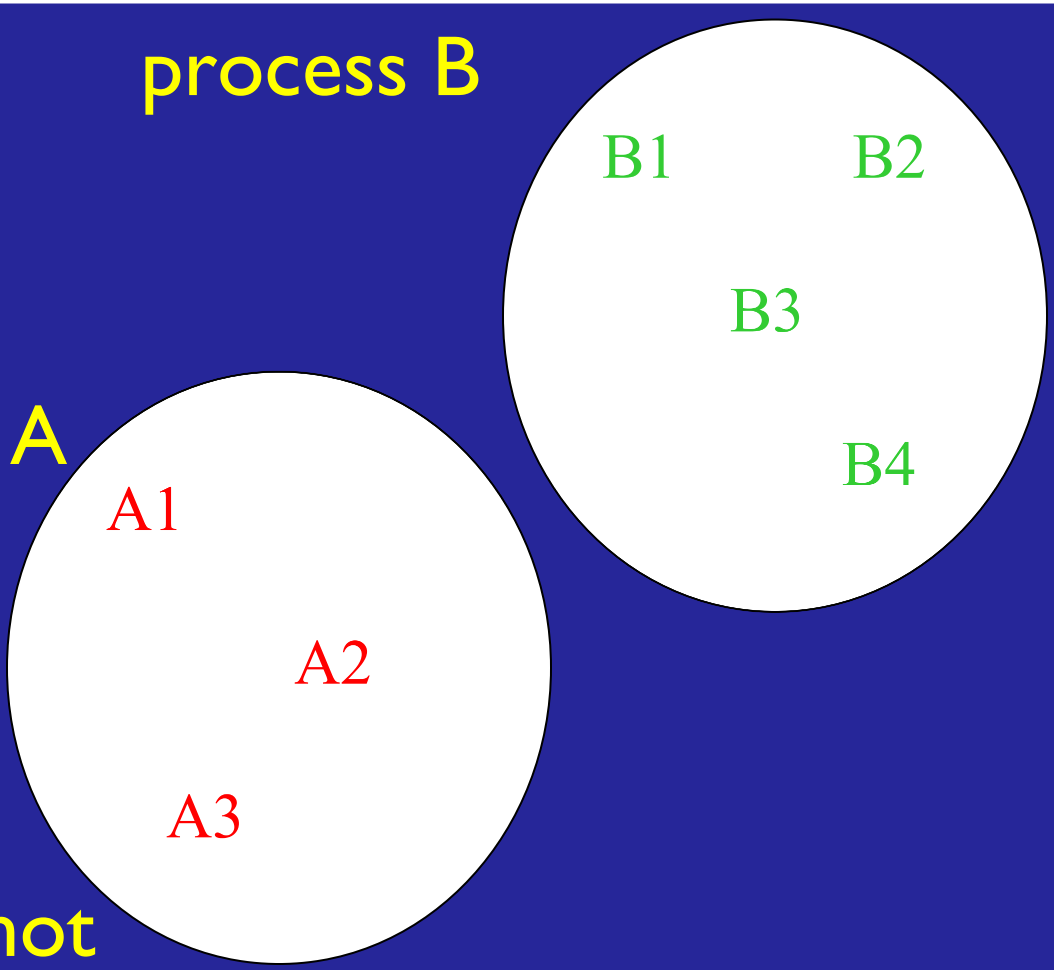




User-space threads run in distinct processes

process B

and process A



have their own

address spaces, and so

threads A2 and B3 cannot

share variables unless you put them in

shared memory segments --- so more details and more coordination are required.

Pros and Cons of user-space threads

- no need to modify OS
- each thread is simple (no OS interrupts, unless time-slicing)
- creation, switching, synchronization is simple (no OS involvement)
- efficient and cheap: switching (simply save/restore thread state),
- management is fairly simple (run-time system),

but

- host process gets one time-slice, no coordination with OS,
- no matter how many threads it has
- and so threads must politely care to share the CPU;
- if one thread blocks on a syscall, the whole process blocks,
- (for example a page-fault in a thread blocks the process),
- so a multithreaded kernel can help them

Kernel-space Threads (e.g., pthreads)

- interface is similar to user-space threads
- but now managed by the kernel, so
- no run-time system required.
- system calls help to create and manage threads inside a process
- OS keeps usual tables to manage processes
- and it can schedule these threads across multiple CPUs
- while they use the same address space

- no such luxury with user-space threads;
- we'd put threads in different processes,
- and these processes had different address-spaces.
- so we resorted to shared-memory segments
- so that threads in two distinct processes could
- see the same variables

Pros and Cons of kernel-space threads

- kernel schedules threads, and so
- processes with threads can be more efficient;
- especially good in applications that often block.
- schedulable threads run in one address space
- so no need to work with shared-memory

but

- because of OS involvement there is cost, and
- they can be inefficient or slow (compared to user-space),
- and usual operations (e.g., context-switch) cost much more
- thread control block is now full and complex
- because the kernel is involved, and so
- both kernel complexity and overheads increase

Why use threads instead of processes?

- applications are naturally multithreaded (concurrent tasks)
  [e.g., browsers, databases, scientific computations]
- threads work in the same address space
- and can be scheduled on multiprocessors.
- They are cheap to create/maintain (stack, registers),
- don't need new address space, global data, other resources
- and context-switching is cheap

Context-switching: save the context (cpu registers, stack pointer) of interrupted thread, retore the context of next thread to run

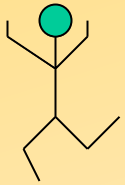
---

The main difficulty: you have a task that can use threads. How can you organize the computation and coordinate the threads?

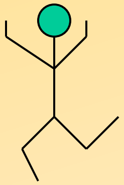
How to think of pthreads

- Bilbo and Frodo are two threads that
- run in one address-space , maybe on different processors,
- can understand and touch the same variables,
- share resources(same file descriptors, heap),
- have independent stack pointers, registers, signals, and
- get their own, scheduled slots with a CPU.
- But here is the rub: say two threads run an "add" function

Bilbo says: "I'll change j to (j+1)"



j = 0



Frodo says: "I'll change j to (j+1)"

So what happens to the value of j at the end?

a race condition

What happens to variable j while they execute?

What happens to variable j when

SIZE is 1 (hmmm …. seems to be doing the right thing)

SIZE is 1000 (still seems okay)

SIZE is 3500 (now we see trouble; why?)

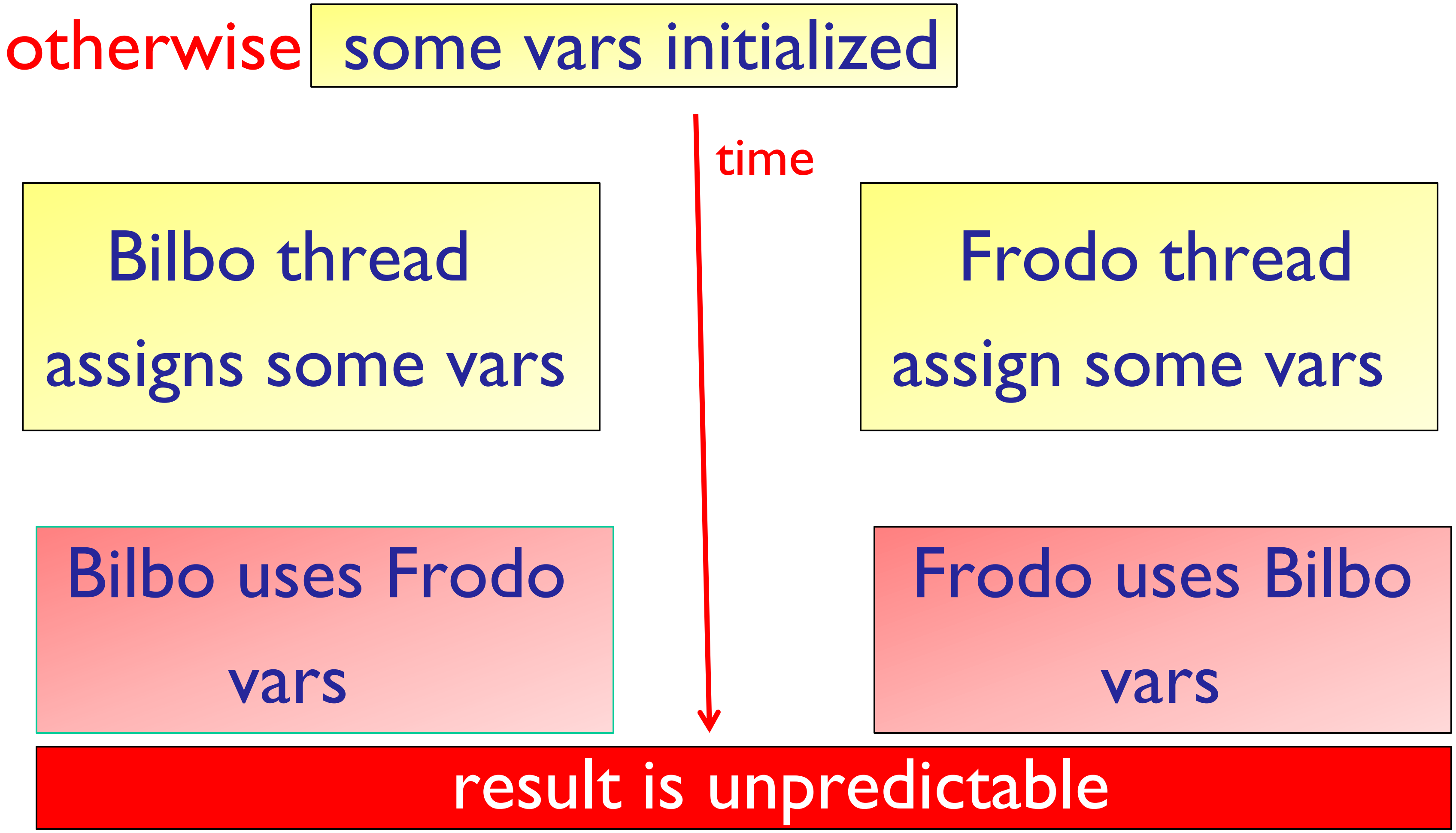
SIZE is 100000 (clear race)

and having separate functions/code does not help

Unix process fork: there is a parent child relationship

Thread_create: no such relationship; threads are "peers," and all on the same footing. Main becomes a thread too.

So we need a way to guard or sequence execution: synchronization



multiprocessor mem models, processor and compiler optimizations will gives you unpredictable execution sequences, along with a race between Bilbo and Frodo to touch variables; very hard to reason about “parallel” or “concurrent” code. So keep things simple.

Terminology and Models

Concurrent programming: tasks occur in any order, and some or all can run at the same time (pthreads!)

Parallel programming: run these concurrent tasks on different processors, all simultaneously (pthreads on a multiprocessor!)

A parallel program is a concurrent program; but not vice-versa

---

Boss-worker model: single boss thread doles work out to workers

Peer-model: no boss, threads operate like a work-crew

Pipeline-model: like an assembly line, each thread does a piece of work and passes it on; when pipe is full, tasks complete while others are part done

Instead of pthreads, let's start with (big) Unix processes that cooperate using a shared memory segment: a simple example

- tasks t1 and t2 both do "something" n times
- n is #defined using TIMES

- "something" is a loop, done BUSYWORK times
  (and BUSYWORK is also #defined)

- t1 and t2 use a shared-mem segment to access two
  integer variables through pointers

- **Processes using a shared-mem segment**