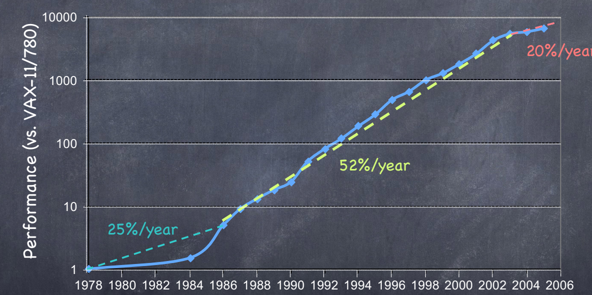
Concurrency and Threads

Uniprocessor Performance not Scaling

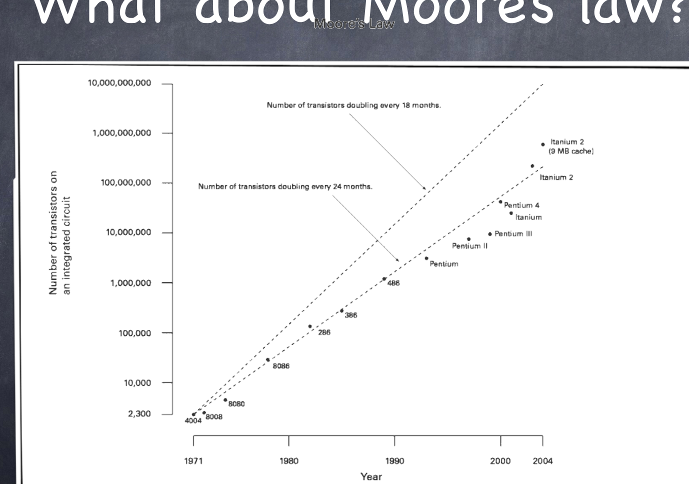


Source: David Patterson

Power and Heat lay waste to CPU makers

- Intel P4 (2000-2007)
  - 1.3GHz to 3.8GHz, 31 stage pipeline
  - "Prescott" in 02/04 was too hot. Needed 5.2GHz to beat 2.6GHz Athlon
- Intel Pentium Core, (2006-)
  - 1.06GHz to 3GHz, 14 stage pipeline
  - Based on mobile (Pentium M) micro-architecture
    - Power efficient
- 2% of electricity in the U.S. feeds computers
  - Doubled in last 5 years

What about Moore's law?



Number of transistors doubles every two years — not performance!

Transistor budget

- We have an increasing glut of transistors
  - (at least for a few more years)
- But we can't use them to make things faster
  - what worked in the 90s blew up heat faster than we can dissipate it
- What to do?
  - make more cores!

Multicore is here - plain and simple

- Raise your hand if your laptop is single core
- Your phone?
- That's what I thought

Multicore Programming: Essential Skill

- Hardware manufacturers betting big on multicore
- Software developers are needed
- Writing concurrent programs is not easy
- You will learn how to do it in this class!

Processes and Threads

- The Process abstraction combines two concepts
  - Concurrency: each process is a sequential execution stream of instructions
  - Protection: Each process defines an address space that identifies what can be touched by the program
- Threads
  - Key idea: decouple concurrency from protection
  - A thread represents a sequential execution stream of instructions
  - A process defines the address space that may be shared by multiple threads

Thread: an abstraction for concurrency

- A single-execution stream of instructions that represents a separately schedulable task
  - OS can run, suspend, resume thread at any time
  - bound to a process
  - Finite Progress Axiom: execution proceeds at some unspecified, non-zero speed
- Virtualizes the processor
  - programs run on machine with an infinite number of processors (hint: not true)
- Allows to specify tasks that should be run concurrently...
  - ...and lets us code each task sequentially

Why threads?

- To express a natural program structure
  - updating the screen, fetching new data, receiving user input
- To exploit multiple processors
  - different threads may be mapped to distinct processors
- To maintain responsiveness
  - splitting commands, spawn threads to do work in the background
- Masking long latency of I/O devices
  - do useful work while waiting

How can they help?

- Consider the following code segment:

```
for (k = 0; k < n; k++)
   a[k] = b[k] × c[k] + d[k] × e[k]
```

- Is there a missed opportunity here?

How can they help?

- Consider a Web server
  - get network message from client
  - get URL data from disk
  - compose response
  - send response

How can they help?

- Consider a Web server

  Create a number of threads, and for each thread do

  - get network message from client
  - get URL data from disk
  - compose response
  - send response

- What did we gain?

Overlapping I/O & Computation

Request 1, Thread 1

- get network message (URL) from client
- get URL from disk

(disk access latency)

- send data over network

Request 2, Thread 2

- get network message (URL) from client
- get URL from disk

(disk access latency)

- send data over network

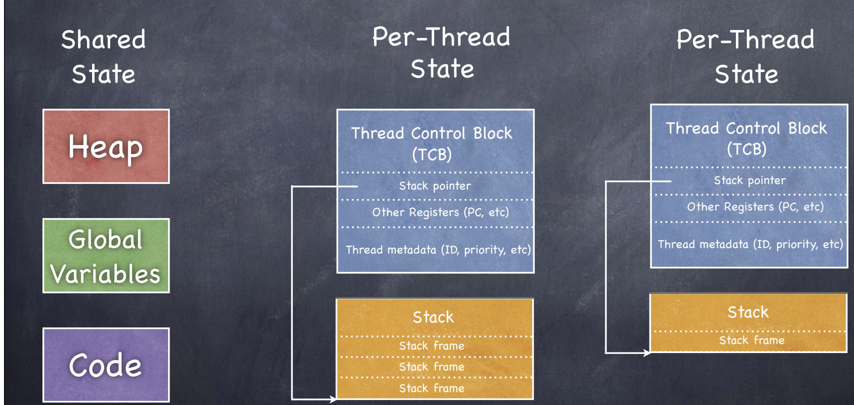
Time

Total time is less than Request 1 + Request 2

Processes vs. Threads

Processes	Threads
Have data/code/heap and other segments	No data segment or heap
Include at least one thread	Needs to live in a process
If a process dies, its resources are reclaimed and its threads die	More than one can be in a process. First calls main.
Interprocess communication via OS and data copying	If a thread dies, its stack is reclaimed
Have own address space, isolated from other processes'	Have own stack and registers, but no isolation from other threads in the same process
Each process can run on a different processor	Inter-thread communication via memory
Expensive creation and context switch	Each thread can run on a different processor
	Inexpensive creation and context switch

Implementing the thread abstraction: the state



Note: No protection enforced at the thread level!

A simple API

`void thread_create(thread, func, arg)`

- creates a new thread in `thread`, which will execute function `func` with arguments `arg`

`void thread_yield()`

- calling thread gives up the processor

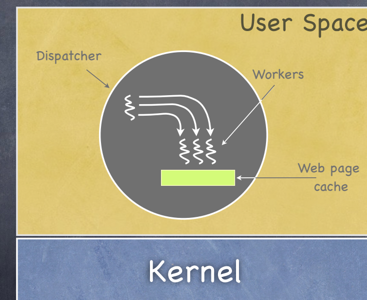
`thread_join(thread)`

- wait for `thread` to finish, then return the value `thread` passed to `sthread_exit`.

`thread_exit(ret)`

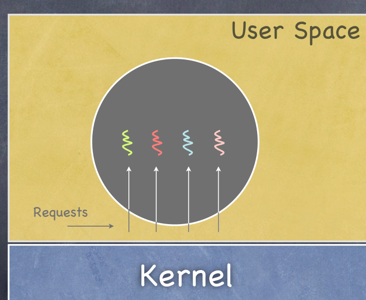
- finish caller; store `ret` in caller's TCB and wake up any thread that invoked `sthread_join(caller)`

Multithreaded Processing Paradigms



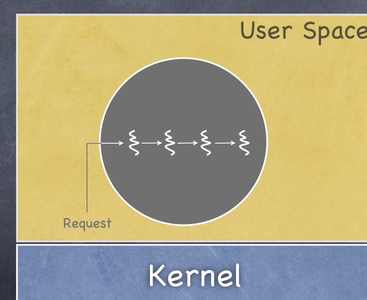
Dispatcher/Workers

Multithreaded Processing Paradigms



Specialists

Multithreaded Processing Paradigms



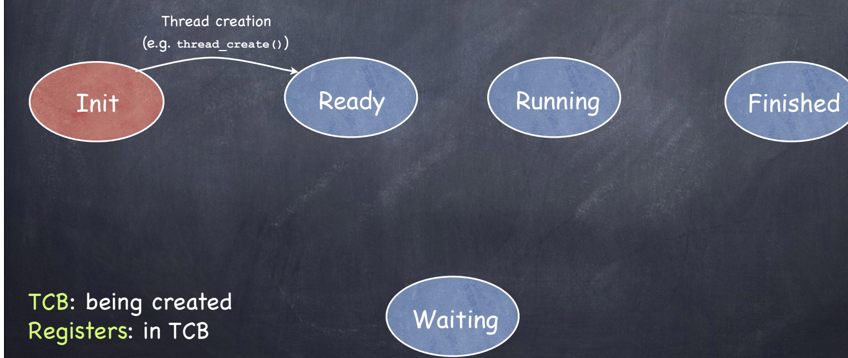
Pipelining

Threads Life Cycle

- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states

Threads Life Cycle

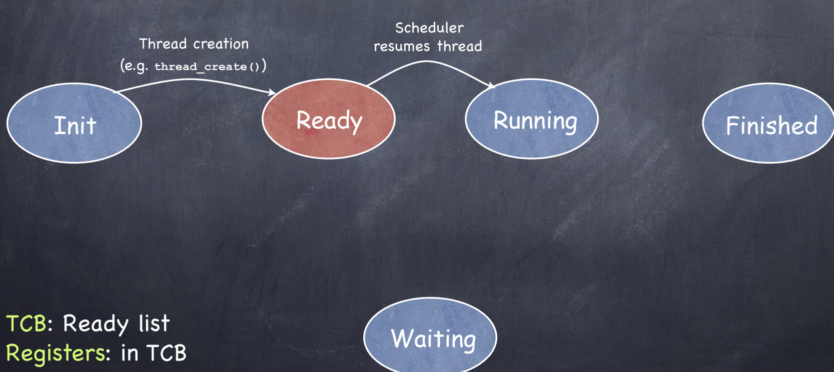
- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states



TCB: being created
Registers: in TCB

Threads Life Cycle

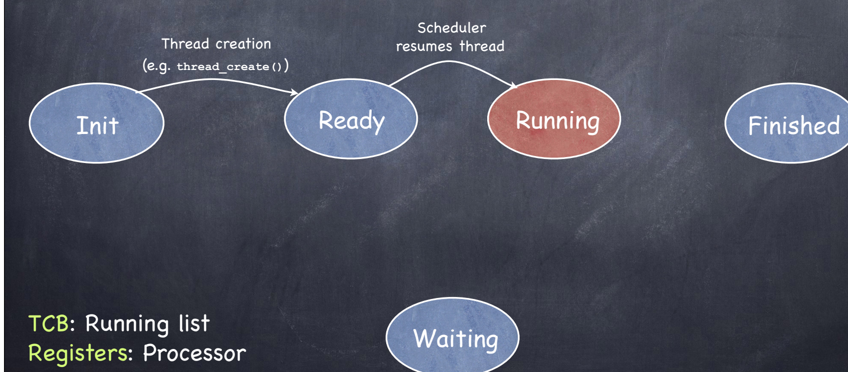
- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states



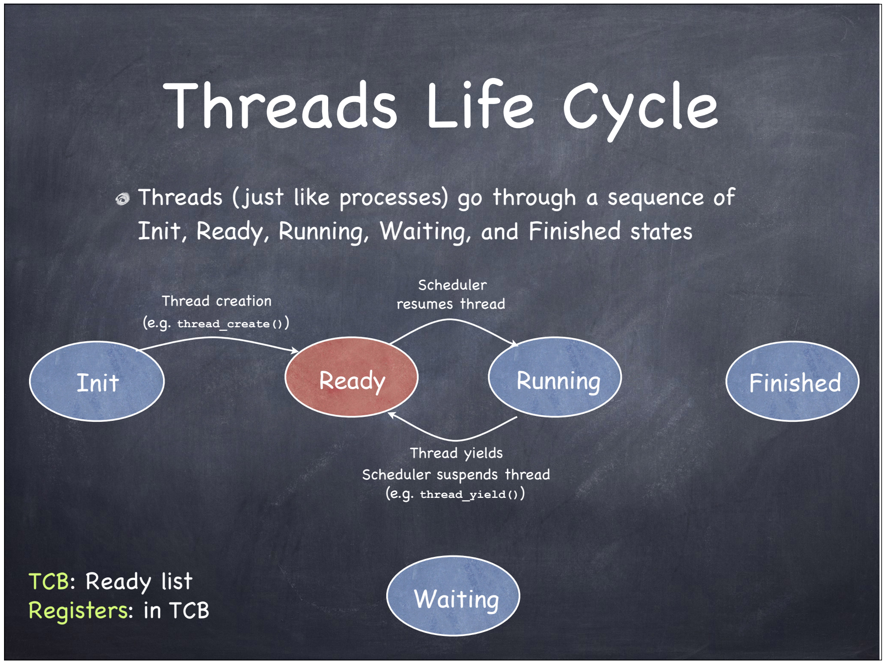
TCB: Ready list
Registers: in TCB

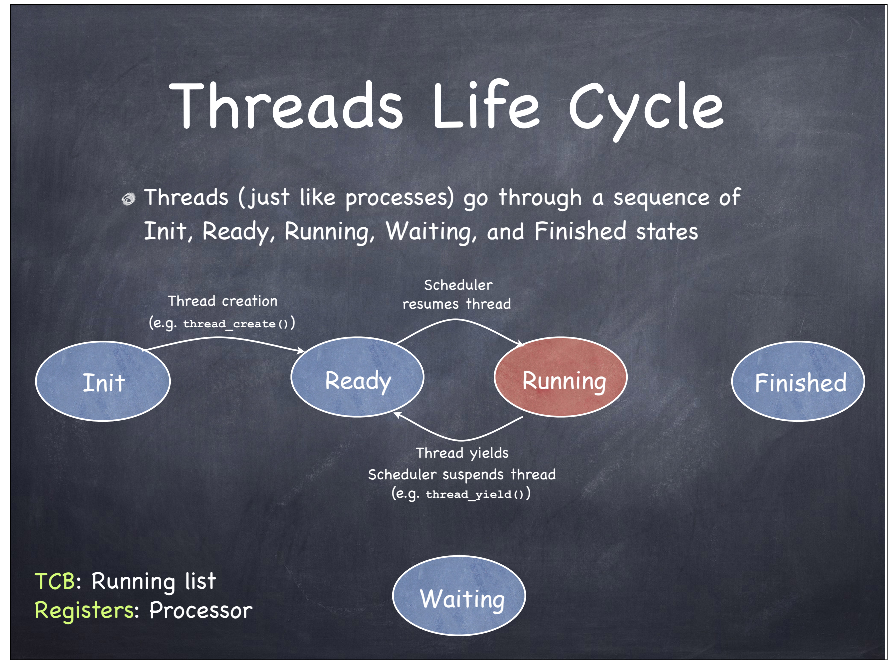
Threads Life Cycle

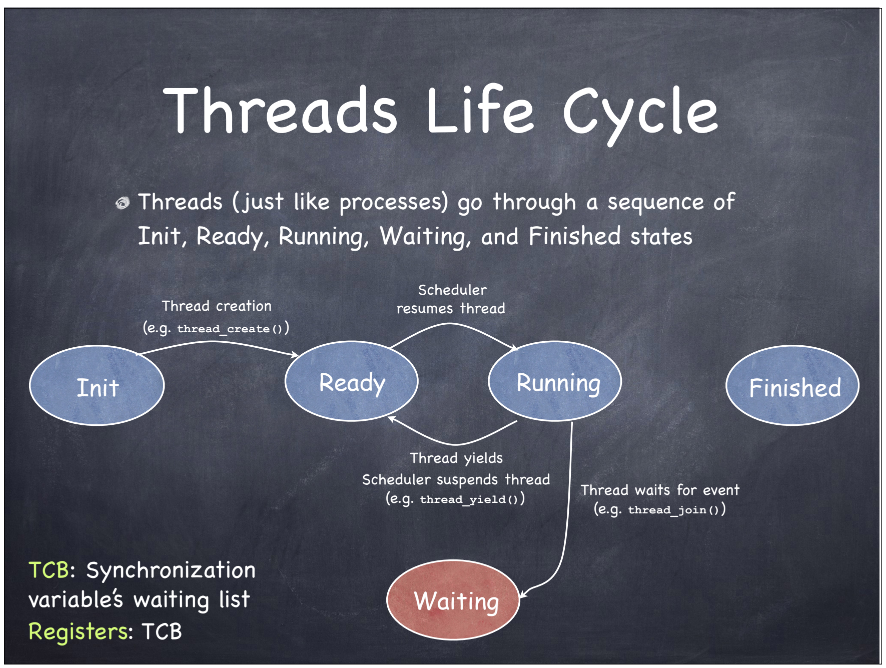
- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states

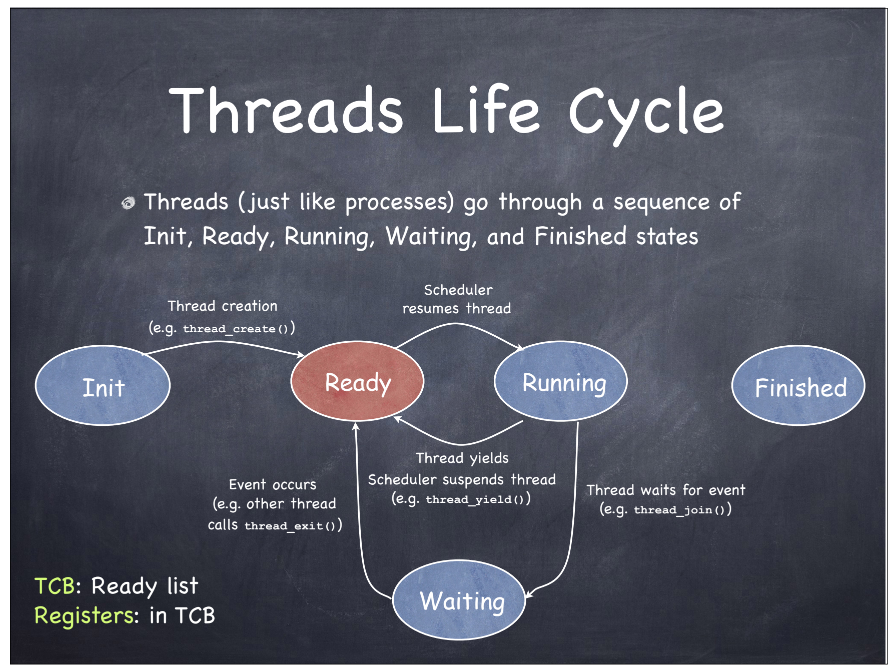


TCB: Running list
Registers: Processor


Threads Life Cycle
Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states
Thread creation
(e.g. thread_create())
Scheduler
resumes thread
Init
Ready
Running
Finished
Thread yields
Scheduler suspends thread
(e.g. thread_yield())
TCB: Ready list
Registers: in TCB
Waiting

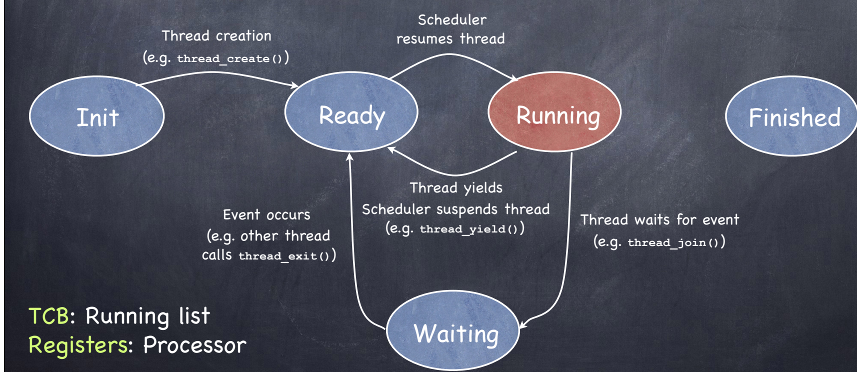

Threads Life Cycle
Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states
Thread creation
(e.g. thread_create())
Scheduler
resumes thread
Init
Ready
Running
Finished
Thread yields
Scheduler suspends thread
(e.g. thread_yield())
TCB: Running list
Registers: Processor
Waiting


Threads Life Cycle
Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states
Thread creation
(e.g. thread_create())
Scheduler
resumes thread
Init
Ready
Running
Finished
Thread yields
Scheduler suspends thread
(e.g. thread_yield())
Thread waits for event
(e.g. thread_join())
TCB: Synchronization variable's waiting list
Registers: TCB
Waiting


Threads Life Cycle
Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states
Thread creation
(e.g. thread_create())
Scheduler
resumes thread
Init
Ready
Running
Finished
Event occurs
(e.g. other thread
calls thread_exit())
Thread yields
Scheduler suspends thread
(e.g. thread_yield())
Thread waits for event
(e.g. thread_join())
TCB: Ready list
Registers: in TCB
Waiting

Threads Life Cycle

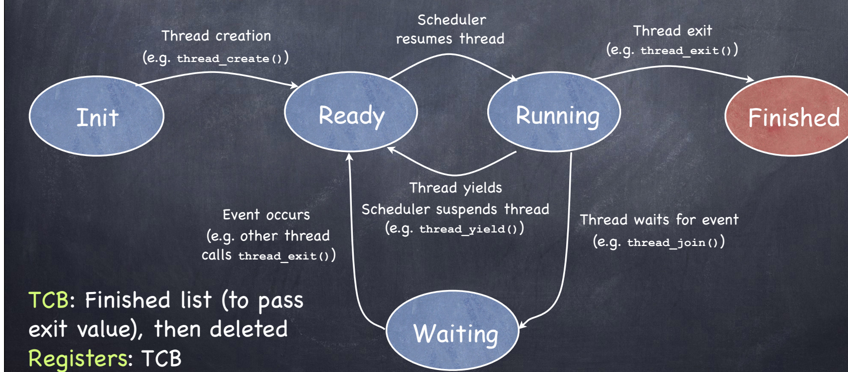
- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states



TCB: Running list
Registers: Processor

Threads Life Cycle

- Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states



TCB: Finished list (to pass exit value), then deleted
Registers: TCB

One abstraction, many flavors

- Kernel-level threads
  - execute kernel code. Common in today's OSs
- Kernel level threads and single-threaded processes
  - system call handlers run concurrently with kernel threads
- Multithreaded processes using kernel threads
  - thread within process make sys calls into kernel
- User-level threads
  - thread ops in user-level library, without informing kernel
  - TCB in user level ready list

Context switching in-kernel threads

- You know the drill:
  - Thread is running
  - Switch to kernel
  - Save thread state (to TCB and stack)
  - Choose new thread to run
  - Load its state (from TCB and stack)
  - Thread is running

Context switching in-kernel threads

- You know the drill:
  - Thread is running
  - Switch to kernel
  - Save thread state (to TCB and stack)
  - Choose new thread to run { Policy decision left to the scheduler
  - Load its state (from TCB and stack)
  - Thread is running

What triggers a context switch?

- Voluntary event
  - via a call to the thread library: `thread_yield(), thread_wait(), thread_exit()`
- Involuntary event
  - e.g., timer or I/O interrupt; processor exception

Voluntary Kernel thread context switch

- Defer interrupts
- Choose next thread to run from ready list
- Switch!
  - save register and stack of current thread in TCB
  - add current thread to ready list
  - switch to new thread's stack
  - slurp in new thread's state from its TCB
  - change state of new thread to RUNNING
- Enable interrupts

One story, two perspectives

System calls: one story, two perspectives

Thread 1 — In-kernel thread's viewpoint

```
while (true) {
   thread_yield()
}
```

"return" from thread_switch into stub
call thread_yield()
choose another thread
call thread_switch()
save T1's state to TCB
load T2's state

return from thread_switch()
call thread_yield()
choose another thread
call thread_switch()
save T2's state to TCB
load T1's state

return from thread_switch()
return thread_yield()
call thread_yield()
choose another thread
call thread_switch()
save T1's state to TCB
load T2's state

return from thread_switch()
return thread_yield()
call thread_yield()
choose another thread
call thread_switch()
save T2's state to TCB
load T1's state

return thread_switch()
return thread_yield()

Thread 2

```
while (true) {
   thread_yield()
}
```

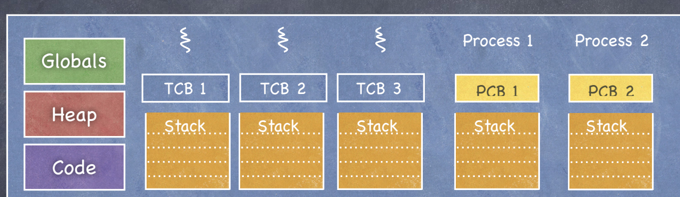
Processor's viewpoint

"return" from thread_switch()
call thread_yield()
choose another thread
call thread_switch()
save T1's state to TCB
load T2's state
return from thread_switch()
call thread_yield()
choose another thread
call thread_switch()
save T2's state to TCB
load T1's state
return from thread_switch()
return thread_yield()
call thread_yield()
choose another thread
call thread_switch()
save T1's state to TCB
load T2's state
return from thread_switch()
return from thread_yield()
call thread_yield()
choose another thread
call thread_switch()
save T2's state to TCB
load T1's state
return thread_switch()
return thread_yield()

Involuntary Kernel thread context switch

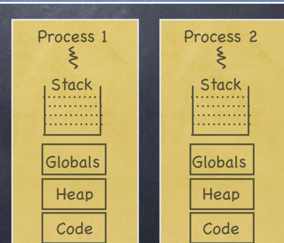
- Save the thread's state in the TCB
  - through a combination of hardware and software
- Run kernel handler
  - can use stack of kernel thread to push variables used by handler
- Restore next ready thread

Single-threaded processes: kernel threads

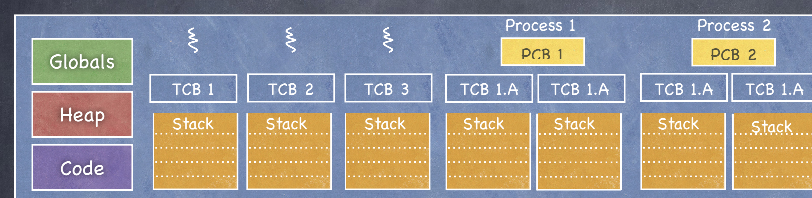


Each kernel thread has its own TCB and its own stack.

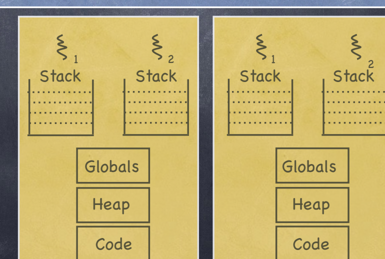
Each user process has a stack at user-level for executing user code and a kernel interrupt stack for executing interrupts and system calls.



Multi-threaded processes: kernel threads



Each user-level thread has a user-level stack and an interrupt stack in the kernel for executing interrupts and system calls.



User-level threads

- No kernel support
- Use upcalls to virtualize interrupts and exceptions
  - TCBs, ready list, finished list, waiting list — in user space
  - thread library calls are just procedure calls!

Fun with concurrency

```
int a = 1, b = 2;
main() {
    CreateThread(&t1, fn1, 4);
    CreateThread(&t2, fn2, 5);
}
fn1(int arg1) {
    if(a) b++;
}
fn2(int arg1) {
    a = arg1;
}
```

What are the value of a and b at the end of execution?

More fun with concurrency

```
int a = 1, b = 2;
main() {
    CreateThread(&t1, fn1, 4);
    CreateThread(&t2, fn2, 5);
}
fn1(int arg1) {
    if(a) b++;
}
fn2(int arg1) {
    a = arg1;
}
```

What are the value of a and b at the end of execution?

Some More Examples

- What are the possible values of x in these cases?

Thread1: x = 1;	Thread2: x = 2;

Initially y = 10;

Thread1: x = y + 1;	Thread2: y = y * 2;

Initially x = 0;

Thread1: x = x + 1;	Thread2: x = x + 2;

This is because ...

- Order of process/thread execution is non-deterministic
  - A system may contain multiple processors and cooperating threads/processes can execute simultaneously
  - Thread/process execution can be interleaved because of time-slicing
- Operations are often not atomic
  - An atomic operation is one that executes to completion without any interruption or failure---it is "all or nothing"
  - x := x+1 is not atomic
    - read x from memory into a register
    - increment register
    - store register back into memory
  - even loads and stores on 64 bit machines are not atomic
- Goal: Ensure correctness under ALL possible interleaving

We have a problem...

- Enumerating all cases is impractical
- We need to
  - define constructs to help with synchronization and coordination
  - develop a programming style that eases the construction of concurrent programs
    - restore modularity
  - more fundamentally, we need to know what we are talking about we we mention "synchronization" or "coordination"...