Today

- Threads
  - vs Processes
  - User vs Kernel threads

Review

- What are some ways that processes communicate?
  - What are the tradeoffs between them?

IPC Tradeoffs Summary

- Communication overhead
  - Higher with message passing, sockets
- Amount of cooperation/collaboration
  - Higher with shared memory, pipes
- Amount of protection
  - Higher with sockets → no direct access

THREADS

Parallel vs Concurrent Execution

Parallel Execution

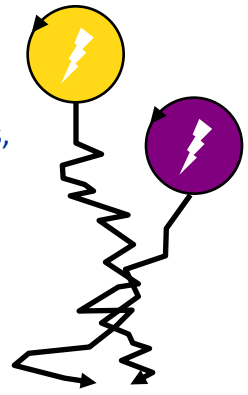
- When two or more execution events are being carried out simultaneously.
- Examples:
  - A Disk I/O and a CPU operation
  - Several CPU operations on a multiprocessor system

Concurrent Execution

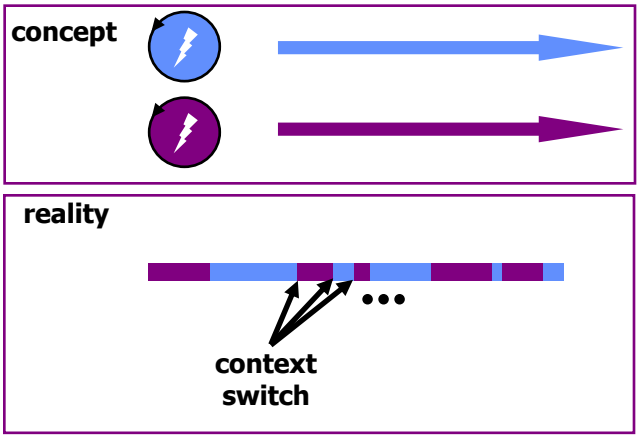
- When two or more execution events either appear to or actually do occur simultaneously.
- A superset of parallel execution

Threads

- A thread is a stream (thread) of control….
  - Executes a sequence of instructions.
  - Thread identity is defined by CPU register context (PC, SP, …, page table base registers, …)
  - Generally, "context" is the register values and referenced memory state
- Multiple threads can execute independently:
  - They can run in parallel on multiple cores...
    - *physical* concurrency
  - …or arbitrarily interleaved on a single core
    - *logical* concurrency



Two threads sharing a CPU



Threads vs Processes

- What experience do you have with using threads (if any)?
  - Why did you use threads?
  - What purposes did they serve?

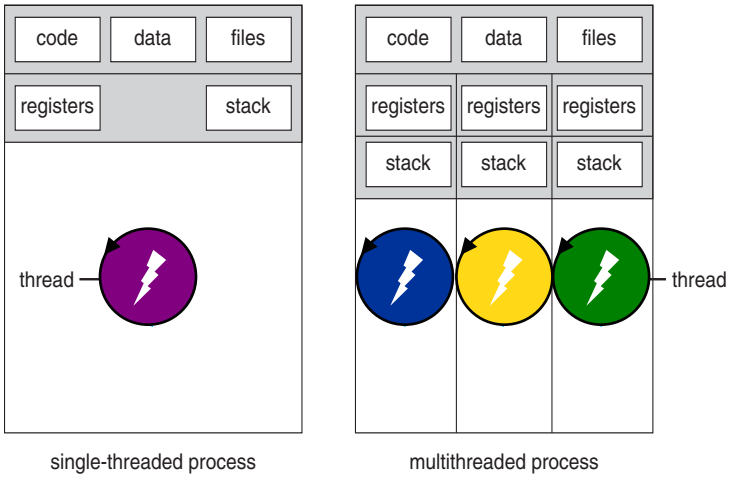
Why use threads?

- Performance
  - Exploiting multiple processors
  - Exploiting multiple I/O devices
  - Scaling
- Responsiveness
  - Long-running tasks run in the background but UI can still respond
- Program structure
  - "This does something different from the main task"
  - "This has some specific [modular] task"

Threads vs Processes

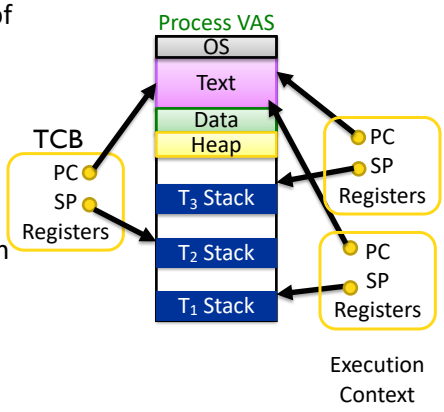
- Threads executing within the same process share *most* of their address space.
- All threads in a process share the same:
  - Code segment
  - Data segment
  - Heap
- Each thread must have its own:
  - Program counter
  - Register values
  - Stack segment (i.e., local variables and parameters)

Single and Multithreaded Processes



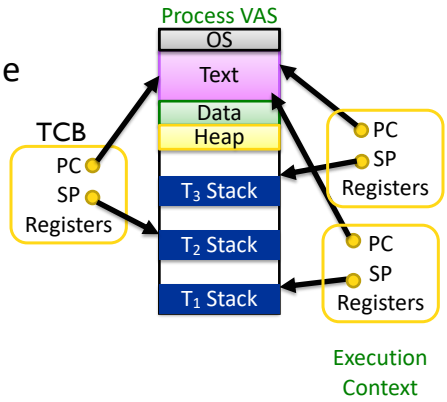
Thread Model: VAS

- Single process with multiple copies of execution resources
- ONE shared virtual address space!
  - All process memory shared by every thread
  - Threads coordinate by sharing variables (typically on heap)
- Recall: Process Control Block (PCB) contains process-specific information
  - Owner, PID, heap pointer, priority, active thread, and pointers to thread information
- Thread Control Block (TCB) contains thread-specific information
  - Stack pointer, PC, thread state (running, …), register values, a pointer to PCB, …



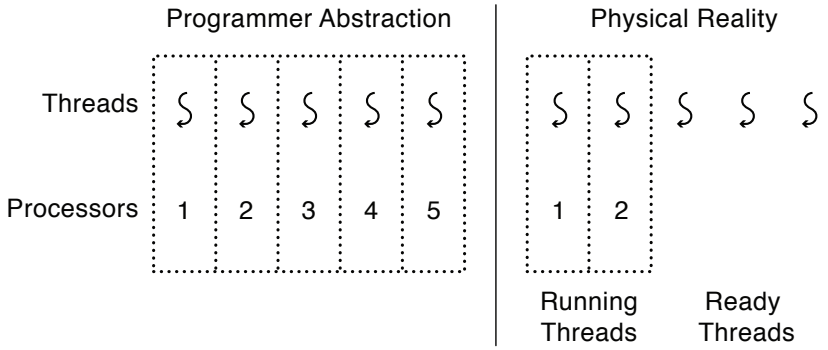
Thread Model: VAS

- Organization of Memory
  - Stacks, heap
- How is this different than the single-threaded process?
  - What were the benefits of that organization?
- What problems does this organization present?
  - How problematic are those problems?

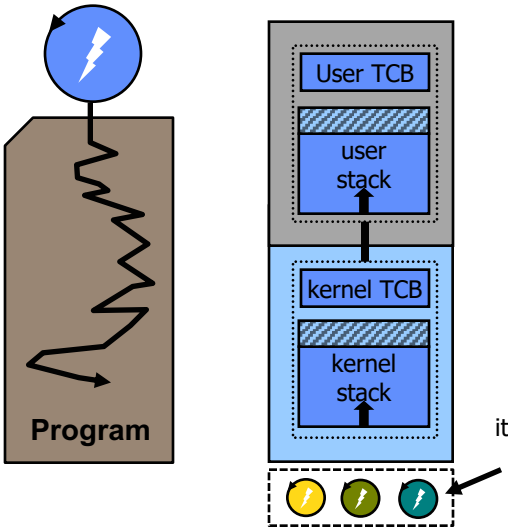


Thread Abstraction

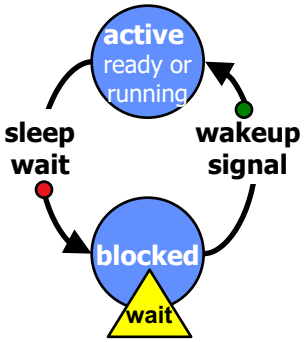
- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with *any* schedule



Thread States



Looks familiar because applies to the ***process*** abstraction too, or, more precisely, to a process's **main thread.**



When a thread is **blocked,** its TCB is placed on a **sleep queue** of threads waiting for a specific wakeup event.

Comparing Threads and Processes

Threads

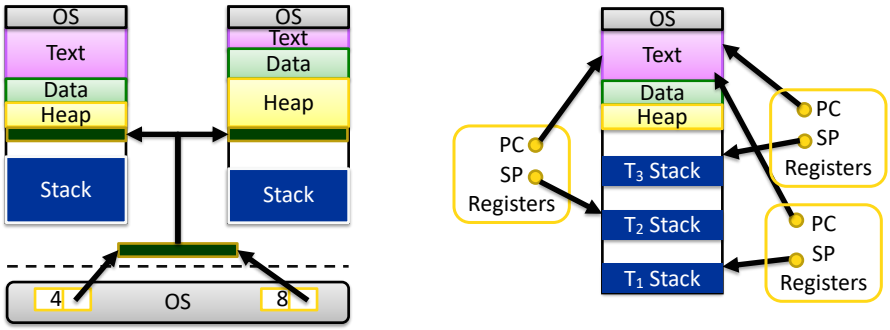
- Has no code or data segment or heap of its own
- Each has its own stack & registers
- Cannot live on its own
  - must live within a process.
  - There can be more than one thread in a process
- If a thread dies, its stack is reclaimed
- Each (kernel) thread can run on a different physical processor
- Inexpensive creation and context switch

Processes

- Has code/data/heap & other segments of its own
  - Also has its own registers
- There must be at least one thread in a process.
- If a process dies, its resources are reclaimed and all threads die
- Each process can run on a different physical processor
- Expensive creation and context switch

Shared Memory vs Threads

- These models are equally powerful (for modern thread libraries)



If interprocess shared memory and threads serve the same roles, why do we prefer threads?

- Threads are easier to use
- Threads provide higher performance
- Users have more control over thread execution/synchronization

Poll Everywhere Start the presentation to see live content. Still no live content? Install the app or get help at PollEv.com/app

If interprocess shared memory and threads serve the same roles, why do we prefer threads?

A. Threads are easier to use.

B. Threads provide higher performance.

C. Users have more control over thread execution / synchronization.

D. Some other reason(s).

Threads vs. Interprocess Shared Memory

- Threads: shared virtual address space → LOW context switch overhead

Why threads are called "lightweight"

- Threads: implicit sharing, no extra calls necessary (opening FDs)
- Multiple processes: more protection
  - ONLY explicitly shared memory is accessible to multiple processes
  - Threads can, for example, overwrite each other's stacks

Looking Ahead

- Project 2 due Monday
- Exam (Wed-Fri)
  - No class Friday