# Chapter 4: Threads & Concurrency

# SANITY CHECKING!!!!!!!

- Process versus thread?
- Race condition?
- Concurrency control?
- Concurrency versus parallelism?

# ADMIN…

- GREAT WORK!!!!
- A1 is AWESOME!
  - Still marking…
- MIDTERM looks GOOD!
  - Marking will take forever, but stay tuned!
- A2 Your first concurrency adventure is posted!
  - NOT using Pthreads YET!
  - It is a bit mind boggling!!!!
  - Understanding what can go wrong and why!!!!!

# (BATTLE)WORM!

- In this part of the assignment, you will implement a simple system for running a series of tasks within one process.
  - Tasks in this system will run separate functions, and your scheduler will need to switch between them.
- These tasks will have the ability to perform *blocking* operations. When a running task blocks, your scheduler will need to stop running the current task and select a new one to run in the meantime.
- You will not use preemption; your scheduler only switches tasks when the currently-running task blocks. This is sometimes referred to as *cooperative scheduling*, since it relies on all the tasks to cooperate by blocking periodically.
- While the system you implement could be used for other purposes, you will be implementing it specifically to support the game **Worm!**, a clone of the classic game *Snake*.
- NOT AT ALL LIKE BATTLESNAKE… but a start!
- DOES NOT USE PTHREADS!!!!!!!

# Details

- This game implementation uses several tasks:
  - A main task that starts up the game
  - A task to update the worm state in the game board
  - A task that redraws the board periodically
  - A task that reads input from the user and processes controls
  - A task that generates "apples" at random locations on the board
  - A task that updates existing "apples" by spinning them and removing them after some time
- Each task runs in a loop that contains a blocking operation.
- Your job is to implement a **scheduling system** that will run these tasks in ***round-robin*** fashion, switching between them as the currently-executing task blocks.
- The starter code has been provided, and take time to look it over as soon as you can!!!

# Assignment 2 Part 1 WORM!

- TRY to get this STARTED RIGHT AWAY!!!!
- TRY to get this working by next week…
- PART 2 (and 3) will be POSTED!

- YOU NEED TO THINK HARD!!!!!!
  - Tasks run until they BLOCK
  - When would each of the tasks
    - Run?
    - Block?
    - End?

# Questions & Answers

- How should task_readchar work?
  - If there's already a character to read, we could return it to the task immediately. However, if *getch()* returns ERR, no input is available.
  - First, you need to record why the task is blocked (it's waiting for input), and then invoke your scheduler to choose a new task to run.
  - Later, when the scheduler is choosing a task to run (at some point in the future) there will be an input character. The scheduler can then *swapcontext()* back to the task we blocked.

# Scheduler System Details

- You will be implementing what is known as *cooperative scheduling*.
  - This is a simple technique for implementing a scheduler without preemption. Once a task is started it will run until it issues a blocking operation or exits.
  - In our case, tasks will run until they exit, wait for another task, sleep for a fixed amount of time, or wait for user input. When you hit one of these points you should invoke the scheduler function to select and start another task to run.
- As part of this process, you will need to check to see if any previously-blocked tasks can now unblock.
  - This could happen because they are
    - waiting for tasks that have now completed,
    - their sleep timer has elapsed,
    - or user input is now available.

# Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models

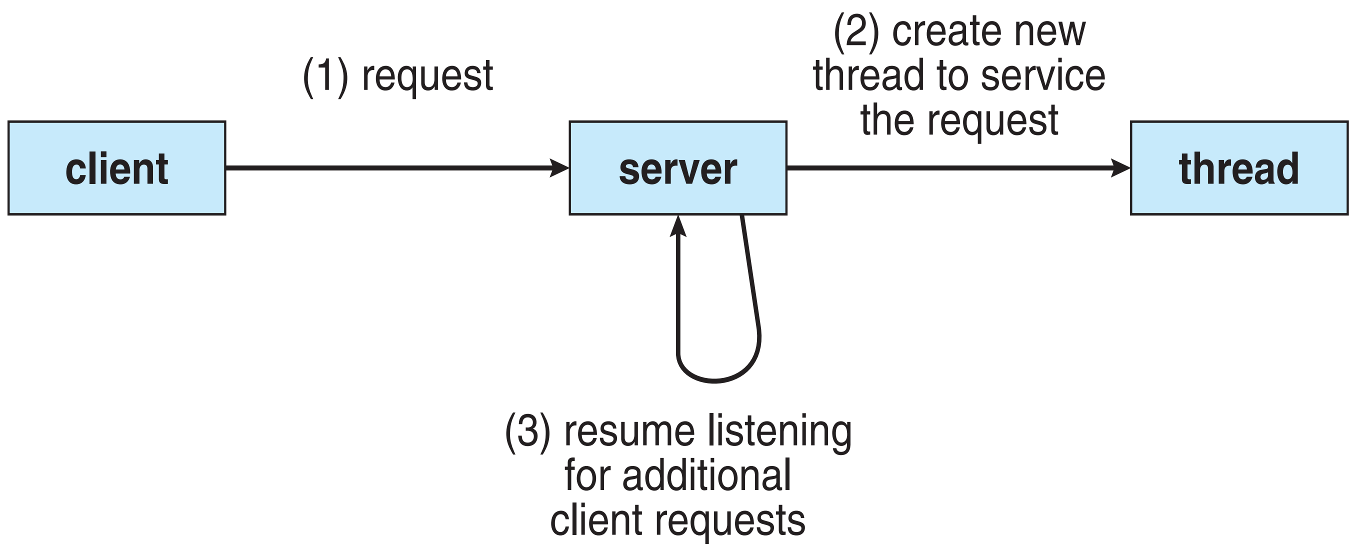
# SANITY CHECKING!!!!!!!

- Process versus thread?
- Race condition?
- Concurrency control?
- Concurrency versus parallelism?

# Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels (NOW!) are generally multithreaded

# Multithreaded Server Architecture



# Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

# Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

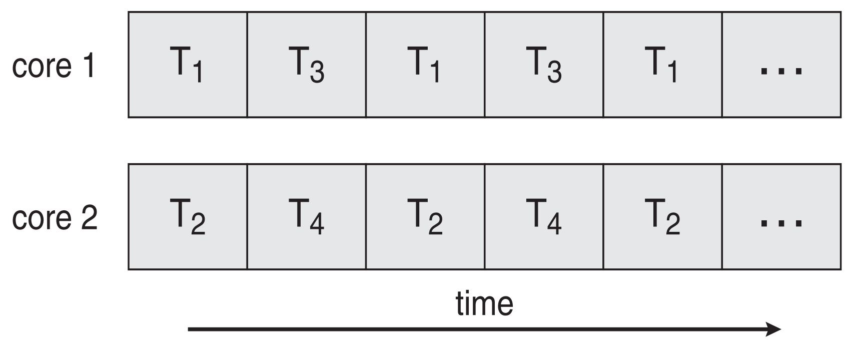
# Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

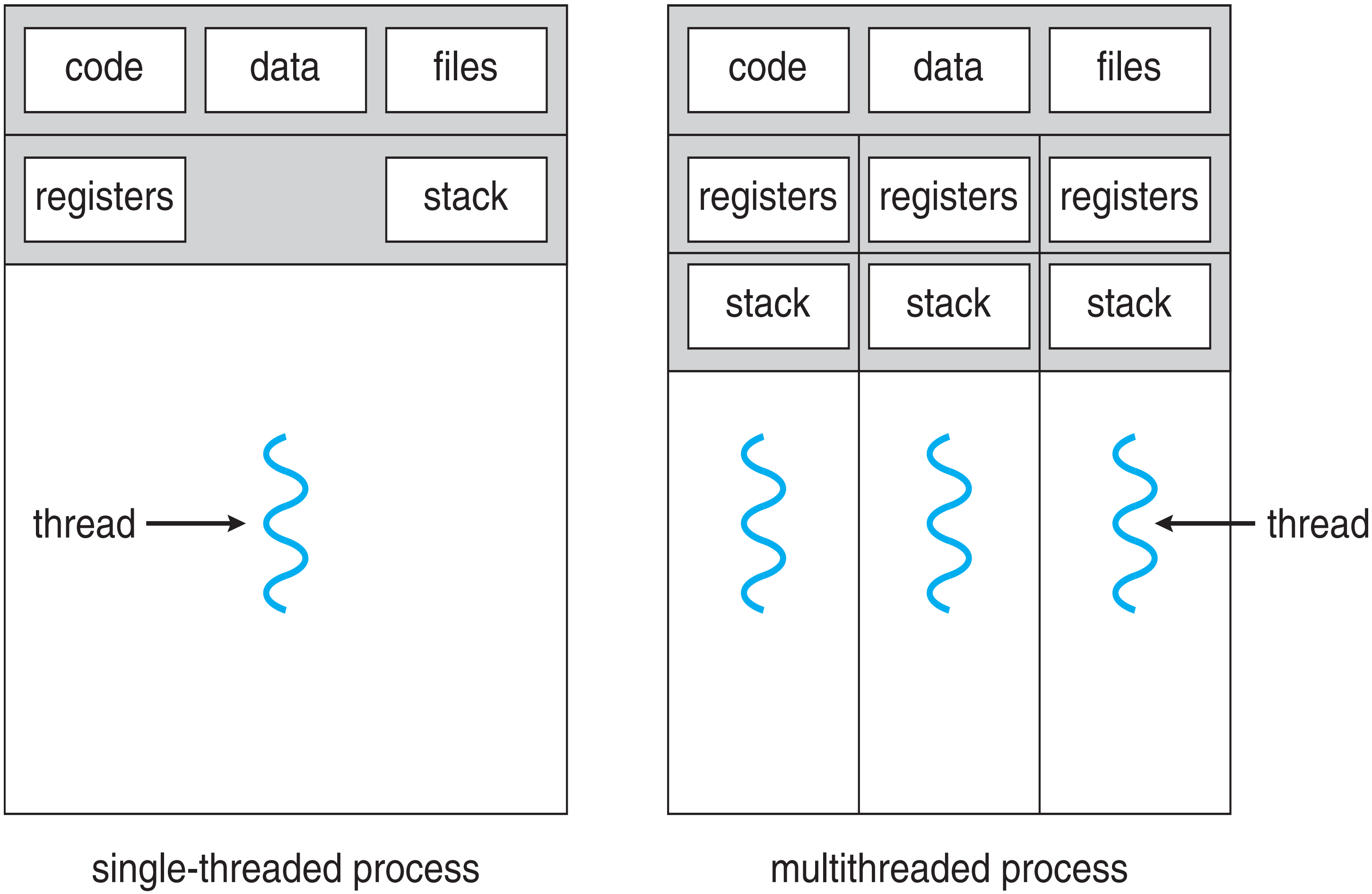
# Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**


- **Parallelism on a multi-core system:**



# Single and Multithreaded Processes



# User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

# Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

# Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

# Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

# BATTLEWORM!

- **Where in the scheduler should tasks actually be executed?**
  Once your scheduler() function has selected a task to run, you call *swapcontext* to switch to that task. When the blocked task resumes, it will look like swapcontext() just returned zero.

- **What are task_exit and the exit context for?**
  The exit context tells the processor where to go when the task's main function returns. The starter code set this up so your tasks will call task_exit when they finish. You should mark the task as DONE and then invoke the scheduler here.

- **How long should test3 take?**
  It should be a total of ten seconds (plus a little random noise)

# BATTLEWORM!

- **When should the scheduler run?**
  When a task calls some function that blocks that task, invoke the scheduler. The scheduler should loop over tasks until it finds one that can run. Once you find the task, *swapcontext* to it. At this point, the scheduler is done.

- **Do you have any hints for task_readchar?**
  If a task blocks waiting for user input, the scheduler will only know if the task can resume if it calls getch(). If getch() returns ERR to the scheduler, the task can't run. If it returns something else, then you can run the task. Whatever value was returned by getch() should then be returned by task_readchar().

# BATTLEWORM!

- **Is a context a list of instructions we need to run?**
  Yes, in combination with the program source and memory. You can think of contexts as a specific spot we can return to in the middle of a set of steps.

- **A hint:**
  If you are switching to a task that is the same as the current task, just return from the scheduler instead of calling swapcontext().