Processes and Threads

Lecture 7

Due Dates

- Finish Lab 2 by the end of this week (Sep 14)
- P1 Checkpoint due by Sep 17
  - Show us your recursive directory listing
  - Finish it early!
  - Have us check you off (there is an entry in the gradebook now)
- P1 tip: work on threads **last**

Today's Schedule

- Pi Environment Setup
- Debugging
- Processes and Threads
- Prizefight: Processes vs. Threads
- Basic Scheduling

Today's Schedule

- **Pi Environment Setup**
- Debugging
- Processes and Threads
- Prizefight: Processes vs. Threads
- Basic Scheduling

Things to Install

- You will want git, a compiler, `make`, and some related tools
  - Shortcut: `pacman -Sy base-devel git`
  - This will install several development packages plus git
- Note the `-Sy`: this tells pacman to refresh its package database
- Bonus: `pacman -Sy sl cowsay cmatrix`
  - (**super** important tools!)

Working Remote

- If you don't want to learn emacs or vim, you have a couple other options to try:
  - Install a nicer console-based text editor. I put a link to several on the schedule page
  - Sync your changes to the Pi
- To sync changes, install Cyberduck (http://cyberduck.io)
  - Configure your editor to be Sublime, VS Code, etc.
  - **Edit** from Cyberduck – changes sync back to Pi!

Today's Schedule

- Pi Environment Setup
- **Debugging**
- Processes and Threads
- Prizefight: Processes vs. Threads
- Basic Scheduling

printf Debugging

- From stackoverflow (https://stackoverflow.com/questions/5765175/macro-to-turn-off-printf-statements):

```
#define LOG(fmt, ...) \
do { \
  if (DEBUG) fprintf(stderr, fmt, __VA_ARGS__); \
} while (0)
```

- An example is on the schedule page!
  - You can define DEBUG using a compiler flag

gdb

- Another helpful resource: using a debugger!
  - If you're on a Mac: lldb
- Compile your program with the -g flag
- Run: gdb ./your-program-name
- Then to run it within gdb:
  - r arg1 arg2 arg3
  - (if no args needed, just type 'r')
- Check out Beej's guide to gdb on the schedule

Today's Schedule

- Pi Environment Setup
- Debugging
- **Processes and Threads**
- Prizefight: Processes vs. Threads
- Basic Scheduling

Processes

- I mentioned in the last class that fork() and exec() aren't the end of the process saga
- On Linux, we can go deeper: **clone()**
  - In fact, fork() is implemented with the clone() system call!

clone

- clone is more flexible than fork: it lets us decide what the child process shares with the parent
- So we can decide whether they share the same heap, open files, etc.
- Here's a cool one: **CLONE_NEWPID**
  - This creates a new process **namespace** with the child process being assigned PID 1
    - And what is PID 1 again, class? Hmm?
    - It's almost like this new process is **contained** in the namespace…

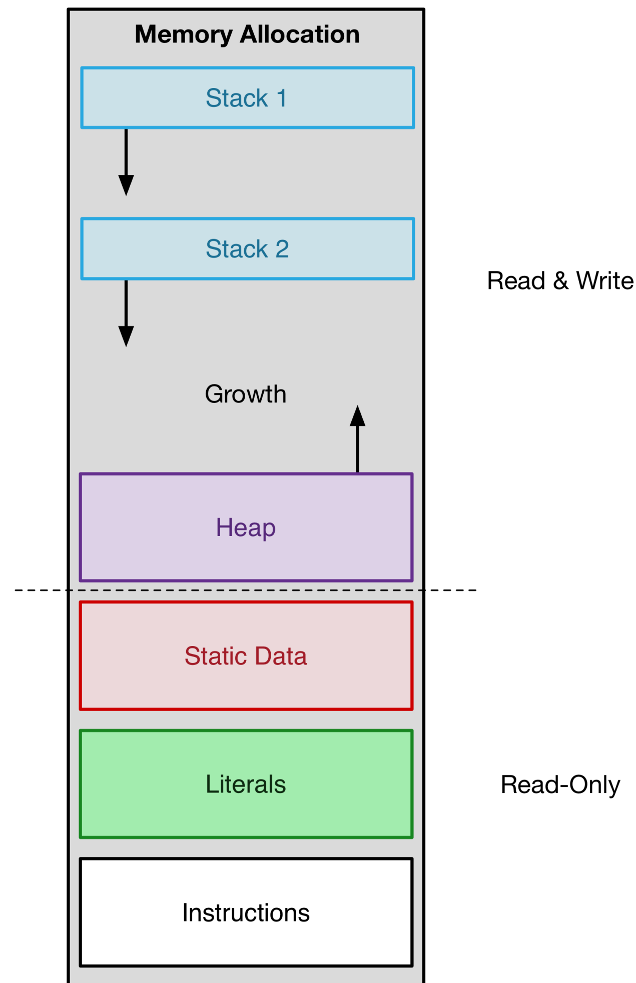
Threads

- As we know, threads are essentially lightweight processes
- Created by processes to do some subset of the work
  - A single process can manage multiple threads
- Threads use **shared memory** for coordination
  - All the threads have access to the program heap
- In our class, we'll use the **pthreads** (POSIX Threads) library to build multithreaded applications

Process Contents

- Each process has:
  - Binary instructions, data, memory
  - File descriptors, permissions
  - Stack, heap, registers
- Threads are very similar, **but** they share almost everything with their **parent** process except for:
  - Stack
  - Registers
  - Program counter

Memory Allocation with Threads



Sharing Data

- Since threads share the heap with their parent process, we can share pointers to memory locations
- A thread can read and write data set up by its parent process
- Sharing these resources also means that it's faster to create threads
  - No need to allocate a new heap, set up permissions, etc.

Creating a Thread – **pthreads**

```
int pthread_create(
        pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine)(void *),
        void *arg);
```

pthread_create

1. pthread_t, is considered an **opaque type**, defined internally by the library
   - It's often just an integer that uniquely identifies the thread
2. pthread attributes
   - Include **stack size** and scheduling policies
3. Function pointer to execute
4. Args (pointer to anything – void *)

pthread_join

- `int pthread_join(pthread_t thread, void **value_ptr);`
- The pthread_join function waits for a pthread to finish execution (by calling **return**)
  - The return value of the thread is stored in **value_ptr**
- Commonly used to coordinate shutting down the threads, waiting for their results, etc
- Similar to **wait()**

pthread_detach

- Normally, threads will continue to live on until they are joined
  - Even if they have exited
- Joining cleans up the resources allocated to the thread (such as its stack)
- Sometimes we can't join threads. By **detaching** them, we tell the OS to clean them up once they exit

pthread_exit

- Any guesses what pthread_exit does?
- You’re right, it launches your default web browser!
  - Note: if you’re skimming these slides to cram for a quiz later, that was only a joke
- pthread_exit has a handy property: if you call it in main(), the main thread won’t exit until the rest of the threads are finished
  - Nice if you want to wait for your workers to finish
  - What happens if you just call exit() or return from main?

Today's Schedule

- Pi Environment Setup
- Debugging
- Processes and Threads
- **Prizefight: Processes vs. Threads**
- Basic Scheduling

Multi-Process or Multi-Thread?

- In general, using multiple processes is simpler and easier to implement
  - Split up the problem, have the processes work on it independently
- However, if the algorithm you are implementing requires lots of communication or shared state, threads are a better choice
  - They are also faster to create and require fewer system resources

Speed Difference: Creation

- On Solaris, creating a process is 30x slower than creating a thread, and context switches are 5x slower
- Each operating system has different overhead associated with processes/threads
  - Windows: **huge** process overhead. Use threads where possible
  - Linux: relatively low process creation overhead
- These speed differentials impact how the OS is built!

Scalability

- Remember: single-threaded (or single-process) applications cannot run on more than one CPU
- This impacts **scalability**: the ability of your algorithm to run faster when given more resources
- Threads are a great way to take advantage of more cores to carry out background tasks and make applications more responsive

Threads on Linux (1/2)

- In Linux land, threads are a bit special
- More specifically, they don't exist
  - That's right, you've been lied to throughout your entire CS education
- At the kernel level, Linux makes no distinction between threads and processes
  - They are all just **tasks** that must be scheduled
    - **Contexts of execution** (COE)
  - Child tasks share a varying level of information with their parents

Threads on Linux (2/2)

- So wait, a thread is a process is a thread is a task?
  - **Yes.**
- So how is pthread_create implemented?
  - clone()
- What about fork?
  - clone()
- There still can be overhead depending on **how much** information must be shared between tasks

Stepping Back

- Are threads actually lightweight processes? Yes!
  - They are literally processes that share some information with their parent
- Does this task abstraction matter? Not usually.
  - We can still think about processes and threads in a traditional way
- Still it's good to know how these actually work when we start to discuss **scheduling…**

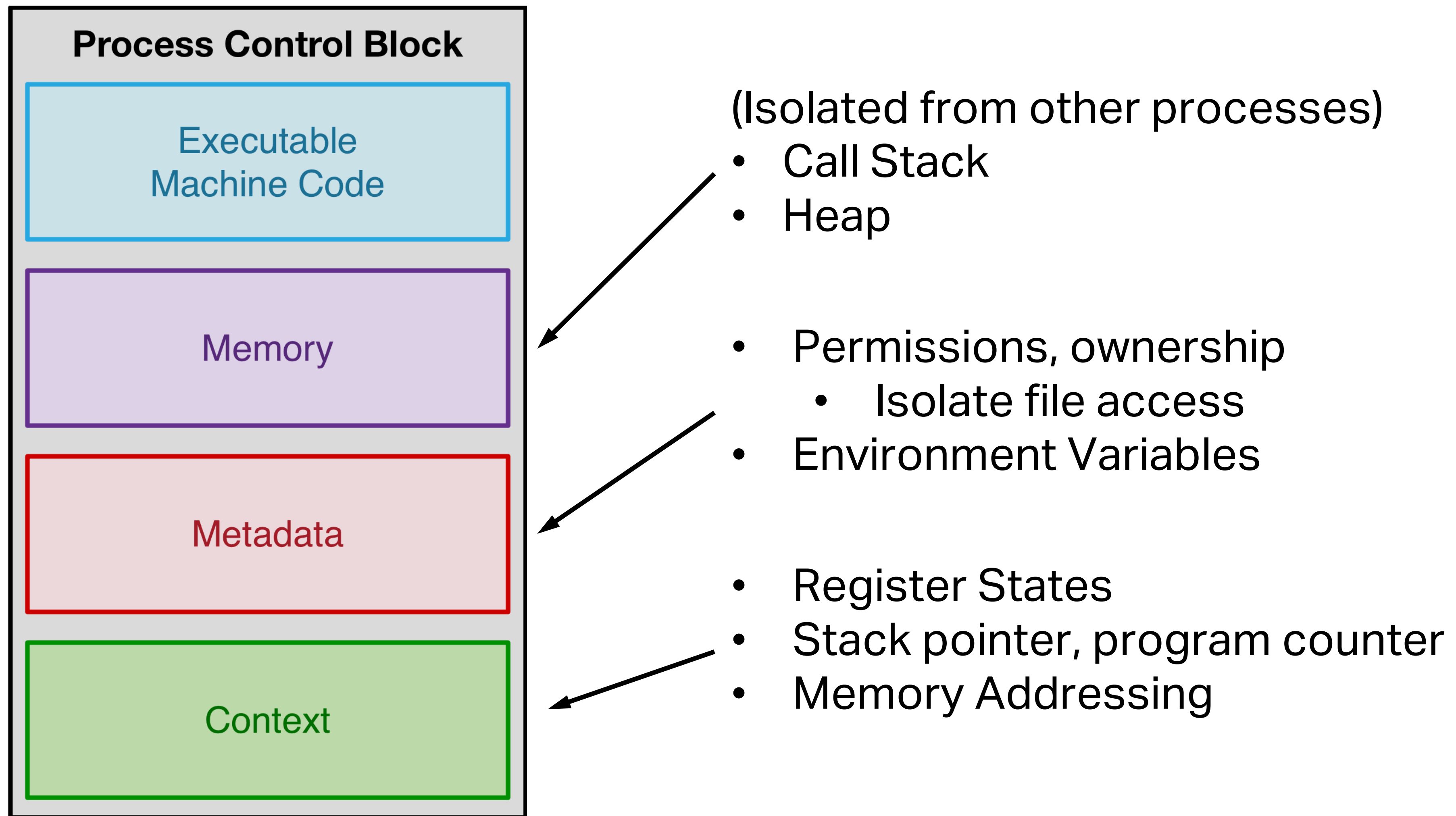
Today's Schedule

- Pi Environment Setup
- Debugging
- Processes and Threads
- Prizefight: Processes vs. Threads
- **Basic Scheduling**

Context Switching

- As discussed previously, switching execution between processes requires a **context switch**
  - Saving and restoring the process control block
- You may wonder if transitioning between user and kernel space requires a context switch
  - In general, no
  - Only a privilege change occurs

Process Control Block



Basic Scheduling

- Vintage OS often used a very basic form of scheduling:
  - Running tasks sequentially (mainframes! punchcards!)
  - Cooperative multitasking
- We can implement our very own user space “thread library” by context switching within our own programs

Demo: Cooperative Multitasking

- See:
  - cooperate1.c
  - cooperate2.c

Interrupt-Based Scheduling

- We mentioned earlier that the OS can **preempt** running processes
- This is accomplished via interrupts
- The OS configures a hardware timer to fire on a set interval:
  - "Interrupt CPU in X ms"

Interrupt Handling

- The trap table is used to determine what to do when the hardware timer fires
- Ultimately moves us back into kernel space
- When this occurs, save the running PCB and replace with the next in the run queue

Advanced Scheduling

- How do we decide what to run next?
- Several **scheduling algorithms** exist, all with different run time properties
- Our next subject: scheduling in depth…