CSci 4061
Introduction to Operating Systems
(Thread-Basics)
Today

• Signal/IPC wrap-up
• Start threads
• Project 3 may be delayed, stay tuned
Threads

- Abstraction: for an executing instruction stream
- Threads exist within a process and share its resources (i.e. memory)
- But, thread has its own stack and “PC”
- Default: always 1 thread (implicit)
Two Threads Sharing a CPU

What may cause a switch?
Thread Benefits

• Concurrency
• Modularity
• Parallelism
• Scale
• Overhead
Threads Example: editor

When one blocks, another can run ...
Thread example: Web server...

When one blocks, another can run ...
How to Quantify the Benefit?

- Web request: read request from network, fetch page, write request to network

\[ T_{\text{req}} = T_{\text{read}} + h \cdot T_{\text{cache}} + (1-h) \cdot T_{\text{disk}} + T_{\text{write}} \]

\[ T_{\text{req}} = T_{\text{read}} + T_{\text{serve}} + T_{\text{write}} \]

- Can we improve the performance of a single request with threads?
  - No, why?

- Can we improve the performance of the “service”, requests/time?
  - Yes!
Example
Example

• Way to think about it:
Thread Example: Web server

dispatcher (...) {

while (TRUE) {
    // 1. read (req ~ URL)
    get_next_request (&req); 
    handoff_work (&req, &buf);
}

worker (...) {

    wait_for_work (&buf, &req)
    // 2. service
    look_for_page_in_cache (&req, &answer);
    if (page_not_in_cache (&answer)) {
        read_page_from_disk (&req, &answer);
        put_page_in_cache (&req, &answer);
    }
    // 3. write
    return (&answer);

• How are these threads interacting?
  • Shared memory: threads share buffer, cache
  • Threads share globals, heap, NOT stack
Looks great

• Drawbacks?

• Alternatives?
Drawbacks

• Sharing
  • Synchronization is needed to protect shared data structures: Web server? Editor?
  • Assume: threads may be switched unpredictably!
  • Failure: no isolation

• Thread-safety (related to Sharing)
  • Not all system calls may be thread-safe
  • System call (or any call) that can be executed concurrently by multiple threads

• Global variables
  • Per thread globals may be needed
Drawbacks: Sharing/Thread-safe

```c
int counter = 0;
int increment_counter() {
    counter ++;  // counter = counter + 1
    return counter;
}
```

problem? Suppose threads T1 and T2 call it

To be thread-safe the shared variable `counter` needs to be protected by a lock
Thread Safety (cont’d)

```c
text
int counter = 0
lock_type counter_lock;
int increment_counter (){  
   // lock is held or free: if held, caller is blocked  
   lock (counter_lock);
   counter ++;
   unlock (counter_lock);
   return counter;
```

Unix man pages will tell you if a syscall is thread-safe...or not
Locks

• Just to be safe, shouldn’t I always put locks around my code?
But ... locks reduce performance

• This is WHY not all system/library calls would be thread-safe: too much overhead to use locks

• So, make sure only 1 thread calls them!
Drawbacks: per thread globals

T1
...
syscall
sets errno
... {T1 switches to T2}
reads errno

T2
...
syscall
sets errno
...

• In Unix, *errno* is a global variable in shared library
• Options to guarantee error reporting is thread-safe?
  • Use locks
  • Eliminate global: return error code
  • Define errno “service” or macro
    
    #define errno _special_thread_errno (thread_id)
Alternatives to Threads

• Want concurrency
  • Goal: If a program (or part of a program) cannot make progress due to blocking, then allow: some other part of the program to make progress

• use processes for concurrency
• poll, non-blocking operations
• signals: event-driven
• Hard to program!
General Thread Options

• Create~Fork
  • allocate memory for stack, perform bookkeeping
  • parent thread creates child threads
  • returns an id

• Destroy/Cancel
  • release memory (or recycle), perform bookkeeping

• Suspend/Yield, Resume

• Wait
  • wait for something, e.g. child finishing
Inside Threads

• A thread contains
  • pc
  • sp
  • registers
  • child threads
  • state

• What about open files?
Threads in Action

- **main** starts executing creates T1 and T2
- **main** blocks at `wait`
- Switches to T1 or T2, say T1
  - blocks at `read`
- Switches to T2

```
main:
create T1, T2
... wait for T1, T2

T1_proc:
... read (...);
...

T2_proc:
...
```
Thread Models

• **Dispatcher-worker (master-slave)**
  - A master process/thread receives request for work
  - Generates/dispatches a thread to service work request
    - E.g. threaded server

• **Two options:**
  - Master can create a thread on “as needed basis” *pop-up*
  - Master can keep a thread pool
    - May reduce perceived latency of creating threads to service request
  - Issues?
    - How many?
    - May be cheaper to create a new thread rather than restore an old one!
Thread Models (cont’d)

• Team
  • a collection of peer threads working on some part of a problem together

• identical threads (parallelism):
  • parallel program running on shared-memory
  • \( n \) threads are created and each are given a share of the problem
    e.g. scale element of a matrix

• different threads (concurrency):
  • editor example
Thought Question

• On a uniprocessor
  • Threaded matrix multiply program
  • NxN matrix (N is large) and sitting in memory
  • Create 4 threads each responsible for ¼ of the matrix multiply operations
  • Time the 4-threaded version and compare with a single threaded version
    • The 4-threaded version does worse---WHY?
Implementing Threads in User Space
Implementing Threads in the Kernel

A threads package managed by the kernel
User vs. Kernel Threads

- User thread advantages
  - no thread system calls! --cheaper
  - more scalable
  - more portable
  - custom control and scheduling
  - blocking is a big problem!
Kernel Threads

• Advantages
  • thread can block: OS can pick another from same process
  • can exploit multiprocessors

• Hyper-threading
  • hardware support for threads!
Hybrid Thread Models

- One-To-One (Linux, WinXY)
  - Each user-level thread maps to a kernel thread
Hybrid Thread Models

- Many-to-One (Java, Pthreads)
  - Many user-level threads mapped to a single kernel thread