Threads

Module 07

Reading

This module covers section 2.2 in the text.

Why Multi-process?

Multi-process programming allows use to write "programs" that can do two or more things at once!

Operating in parallel is a powerful model, but process are considered a **heavy-weight** approach.

Heavyweight?

Process have their own:

- Address space
- Global variables
- Code (separate copy)
- State (Program Counter, Registers, Stack)
- Open Files
- Accounting

- Creating Processes is expensive
- Communicating between processes is expensive

When are processes a good idea?

- When process creation and communication are *relatively* infrequent.
- When the jobs of the processes are largely independent.
- Multi-process systems can employ different languages, and be moved across networks more easily

Lightweight Alternative

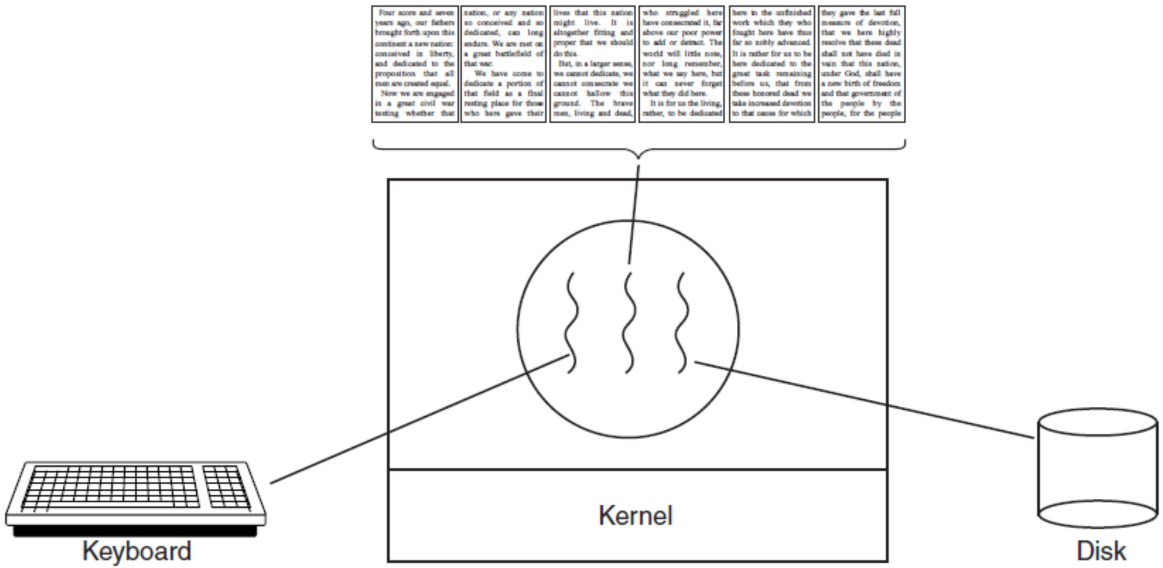
- Often we want to do things in parallel, but don't want to pay the price of multiple processes.
- Threads:
  - Shared Address space
  - Shared code
  - Shared files, accounting, etc.
  - Independent State - PC, Registers, Stack

Threads

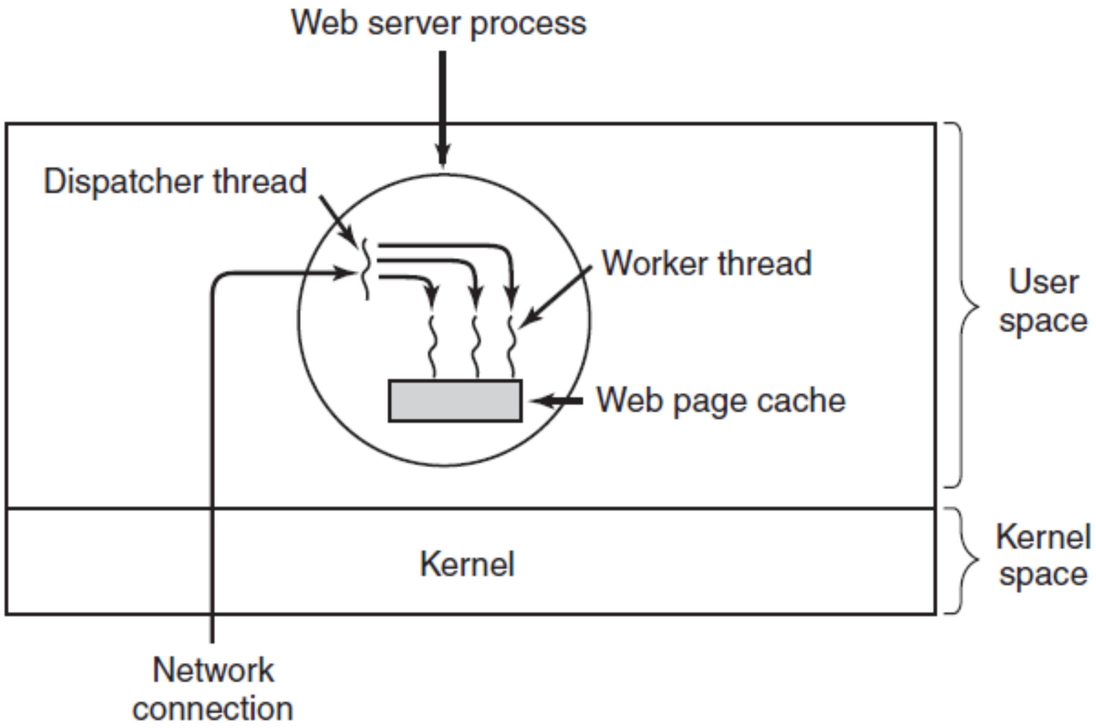
One process can contain many threads

- Each thread has its own execution context - it can be at a different place in the code
- Threads share global data and the **heap** - which facilitates faster (not necessarily easier) communication
- Quick to start

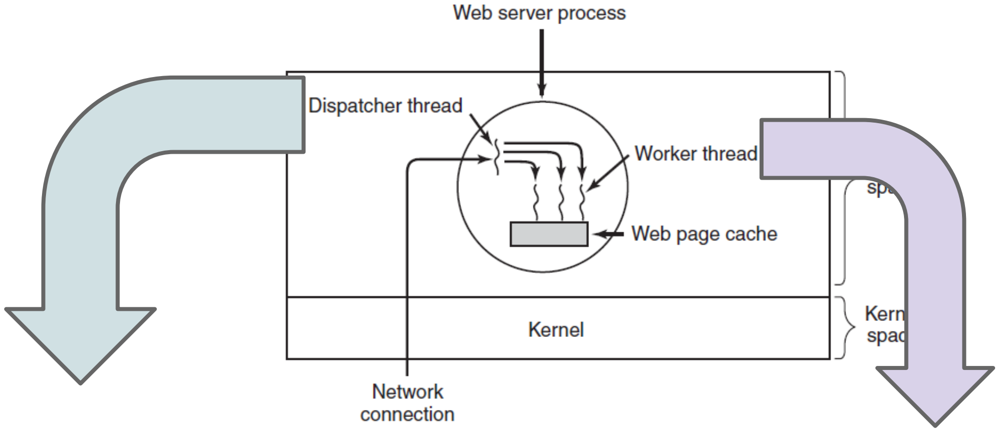
Thread Examples - Word Processor



Thread Example - Web Server



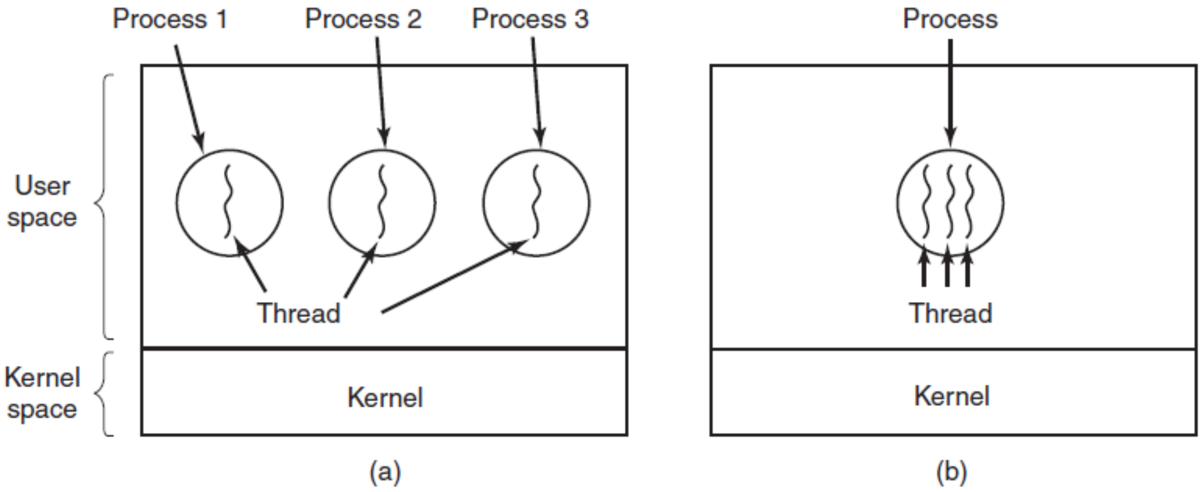
Thread Example - Web Server




```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

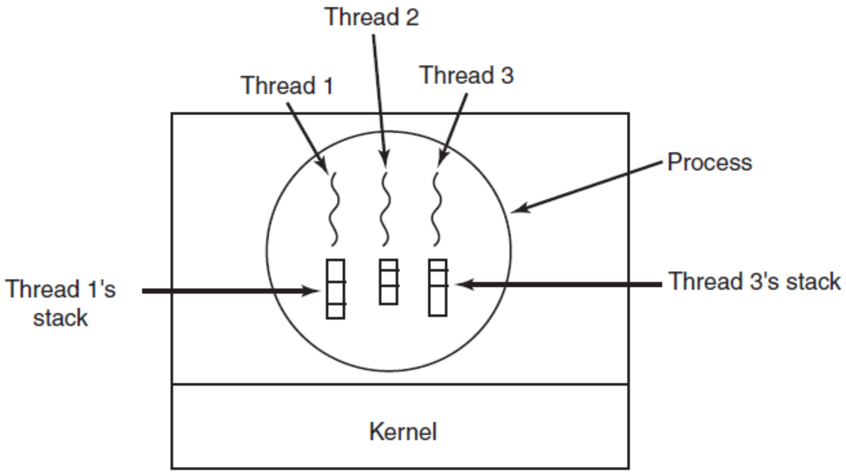
Classical Thread Model



A process *contains* at least one thread

Classical Thread Model

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	



Programming with Threads

- In POSIX, there is a set of standardized API functions for managing threads - called **pthreads**
- On Windows (and we'll do this now), Win32 API provides a very strong API for threading
- C++ 11 also created a thread API that allows you to write portable thread code - we'll see this too.
- But first - we'll see that they all require a standard concept - ***function pointers***

“Review” of Function Pointers

In both POSIX and Win32, creating a thread requires you to provide a function pointer

- This defines where the new thread will start running.
- Just like variables, functions have “types”
- Type equivalent to function signature
- The function’s name is a pointer to “code” instead of data

Function Pointers in C

```c
// Two function, both with the same signature
int add(int a, int b);
int sub(int a, int b);
void print(int a, int b);

// Function that takes a and b and performs the operation
// by executing the function provided:
int execute(int a, int b, int (*function)(int, int) ) {
    return function(a, b);
}
// a call to the execute function:
execute (7, 6, sub);
execute(7, 6, print);  // print doesn’t have the required signature
```

pthread library

Must include phtread.h

Pthreads start in a user defined function with specific signature:

```
void * function_name(void * param)
```

Functions:

- `pthread_attr_init` (set thread attributes to defaults)
- `pthread_create` (create thread)
- `pthread_join(id)` (wait for thread (id) to terminate

compile with -lpthread option

```
g++ -o myprog myprog.cpp -lpthread
```

pthread example

Let’s take a look at `simple-thread.c` for an example

Parallel Example

Now lets take a look at simulating parallel work with posix - thread-summary-posix.cpp

Note - I'm switching over to C++ for a bit!

Windows!

We've neglected Windows so far! *Poor Windows*.

- POSIX uses integers to represent most operating system resources:
  - pid (process id) from fork
  - fd (file descriptor) fopen, pipe
  - tid (thread id) from pthreads
- Windows uses **`HANDLE`** data structures

Threads on Windows

```
HANDLE CreateThread(...)
```

- security attributes (NULL)
- default Stack Size (0)
- thread function
- parameters to thread function
- creation flags (0)
- &thread identifier (DWORD, not really used)

Thread function has strange signature:

```
DWORD WINAPI functionName(LPVOID parameters)
```

Waiting for Threads to stop

- For pthreads, the term is “join” - we join on a thread which means “wait until thread is finished”
- On Windows, there’s a pretty logical set of functions for this:

```
WaitForSingleObject(ThreadHandle, milliseconds)
WaitForMultipleObjects(count, ThreadHandles, all, milliseconds)
```

Windows Example

Let’s rewrite the same summation example, highlighting the Win32 API versus POSIX

`thread-summation-win32.cpp`

C++ 11 Standard

Modern applications almost *always* use threads - and most languages that have *runtimes* provide their own API

- Java Threads call corresponding system calls on target platform - OS independent code!
  - Same with C#, Python, Ruby as well...
- Until recently, C++ didn't - *and this made people sad.*

C++ 11 Threads

C++ Threads are **objects**. This is pretty standard in most OO languages

`std::thread`

The constructor accepts a function, and immediately starts a new thread

```cpp
// thread example
#include <iostream>       // std::cout
#include <thread>         // std::thread

void foo()
{
  // do stuff...
}

void bar(int x)
{
  // do stuff...
}

int main()
{
  std::thread first (foo);
  std::thread second (bar,0);
  std::cout << "main, foo and bar now execute concurrently\n";

  // synchronize threads:
  first.join();     // pauses until first finishes
  second.join();    // pauses until second finishes

  std::cout << "foo and bar completed.\n";

  return 0;
}
```

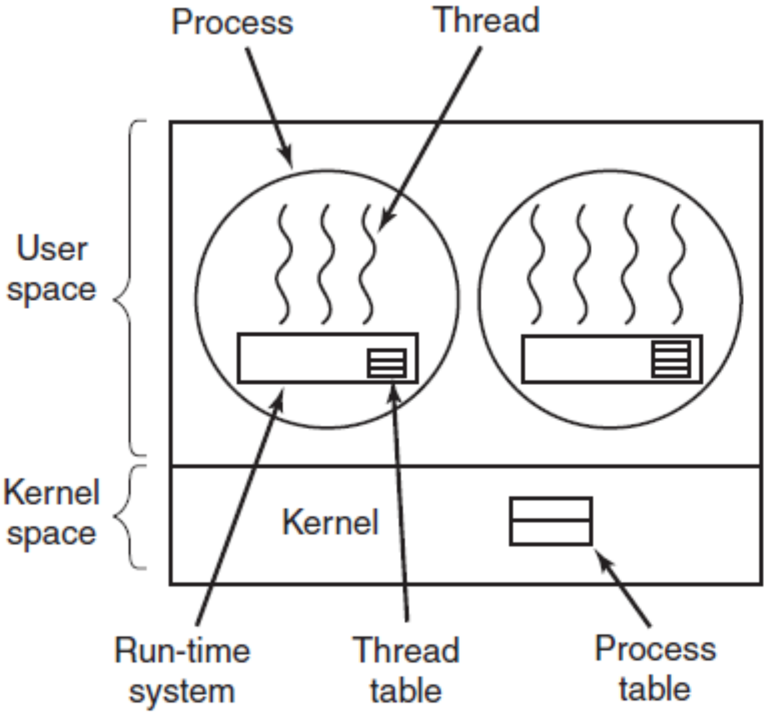
C++ 11 Threads

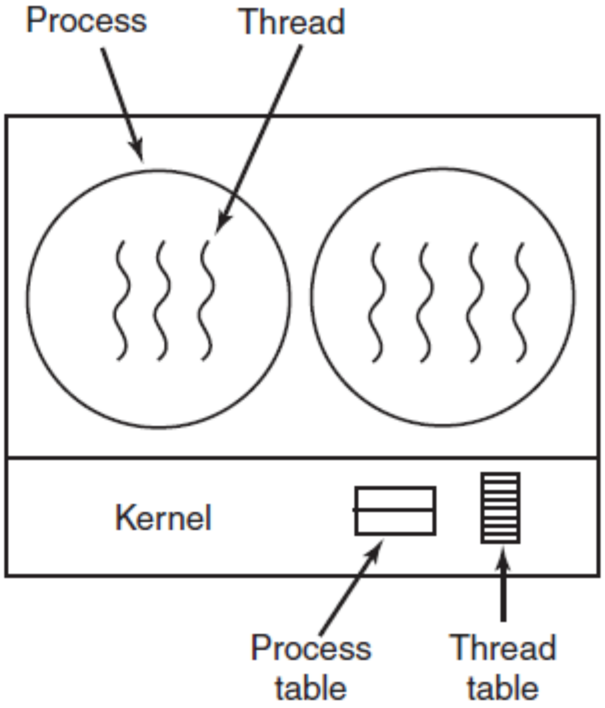
Lets look at `cpp11threads.cpp`

You *probably* need to tell your g++ compiler to use C++ 11 features - this will eventually go away

```
g++ -o test cpp11threads.cpp -std=c++11
```

Kernel vs. User Threads





Kernel vs. User Threads

- Kernel threads are visible to the OS, independently schedulable
  - When one thread is blocked, other threads *in same process* can still be run
  - Multiple threads within same process can run on multiple CPU's
- User threads are invisible to the OS
  - One user thread blocks, all the process's user threads are blocked.
  - Cannot run user threads on multiple CPU's

Kernel vs. User Threads

- User threads are fast, but do not offer the same degree of parallelism as kernel threads
- User threads provide the programming model - but not really the performance benefits
- Nearly all modern operating systems support Kernel Threads

Next up...

After the exam, we’ll start with Chapter 2.3 - Interprocess (and Inter-Thread) communication

This will be **the most** important portion of the semester for your programming skills.