Reading Reference:

Textbook1: Chapter 4 (Section 4.1-4.4)

CONCURRENCY AND THREADS

Tanzir Ahmed

CSCE 313 Spring 2021

Why Processes & Threads?

Goals:

- **Multiprogramming:** Run multiple applications concurrently
- **Protection:** Don’t want a bad application to crash system!

Solution:

- Unit of execution and al, give process illusion it owns machine (i.e., CPU, Memory, and IO device multiplexing)

Challenge:

- Process creation & switching expensive
- Need concurrency within same app (e.g., web server)

Solution:

Thread: Decouple allocation and execution

- Run multiple threads within same process

Motivation for Threads

- Process Context Switch has huge overhead. Requires switching:
  - *CPU state (PC, PSW, all registers etc) - fast*
  - *Memory Address Space - SLOW*
- Assumption: memory access without cache is slow
- A Memory Address Space comes with:
  - *Virtual Memory System data*
    - Page tables (mapping) and Translation Look-aside Buffer (TLB) (i.e., working as cache for mapping)
  - *Actual Memory Content*
    - Dirty (modified) pages that are updated only in cache, not in memory
    - Need to be updated in the memory before switch
- All of these become obsolete after a switch
  - *All caches (L1, L2, ..), TLB become **COLD***
  - *Require time to **WARM UP** as the new process runs*

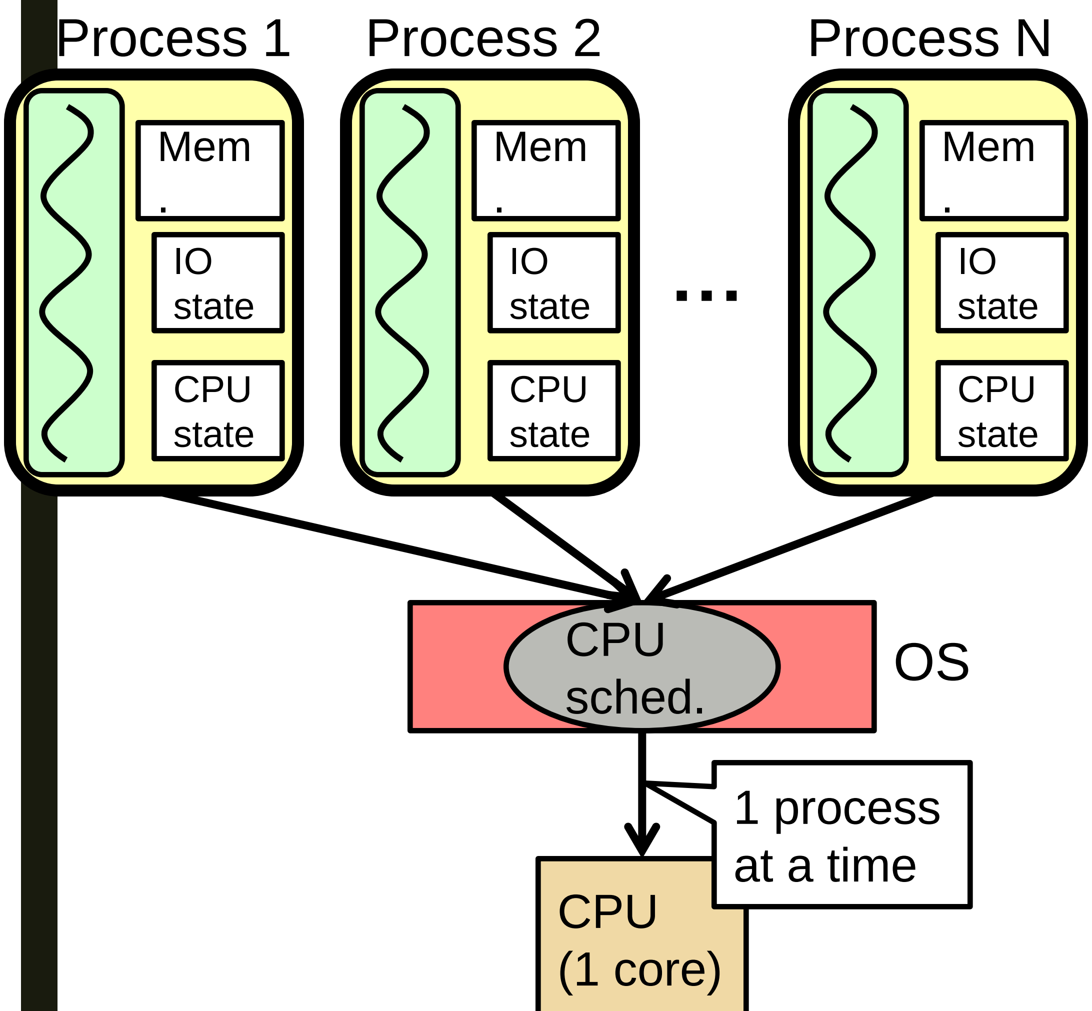
Thread Context Switch

- Still requires:
  - *The kernel scheduler to run, that overhead stays*
  - *Save the current CPU context, and load the next one*
- But, the old address space stays. Thus, no **WARM UP** required for:
  - *Caches (L1, L2, L3, ...)*
  - *TLB (cache for memory mapping)*

Saves time

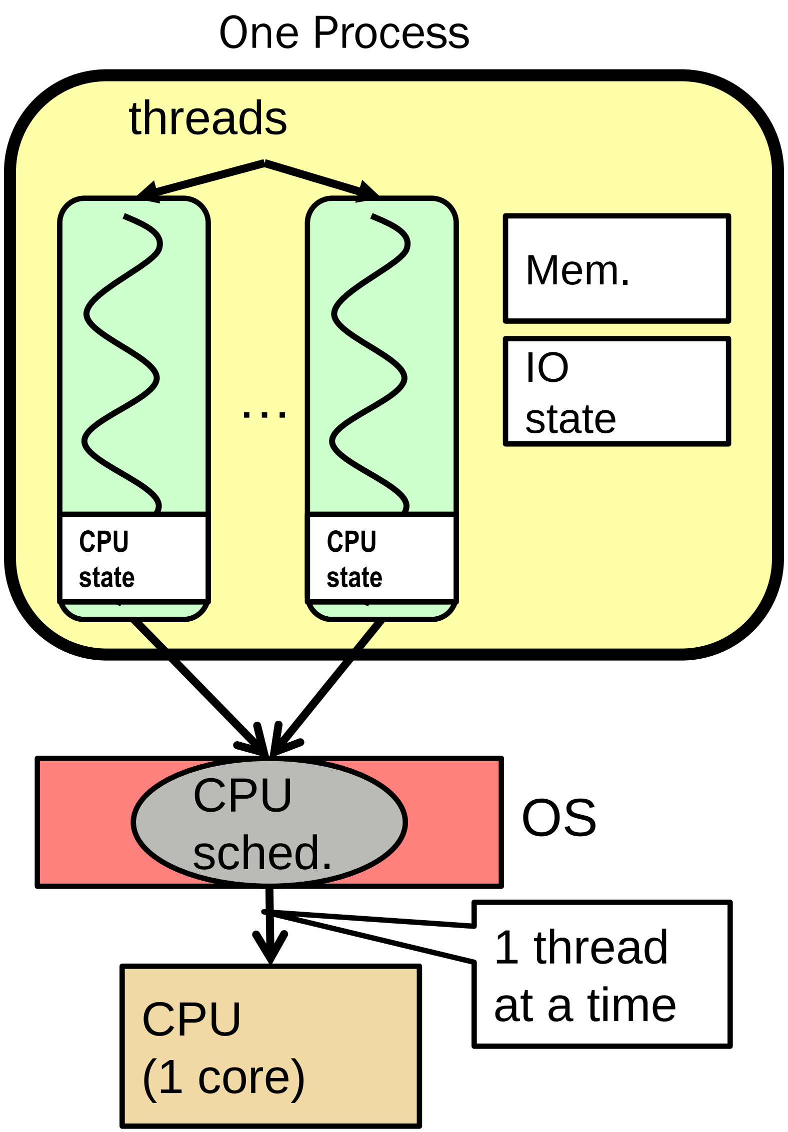
- **Result: Thread Context Switch is much Faster than Process Context Switch**

Processes Context Switch



- **Process** Switch overhead: high
  - *CPU state: low*
  - *Memory/IO state: high*
- Process creation: high
- Protection
  - *CPU: **yes***
  - *Memory/IO: **yes***
- Sharing overhead: high (involves at least a context switch)

Threads Context Switch



- **Thread** Switch overhead: **low**
  - *Only CPU state switched*
- Thread creation: **low**
- Protection
  - *CPU: **yes***
  - *Memory/IO: No*
- Sharing overhead: **low**
  - *No context switches*
  - *Only low-overhead thread-switches*

Threads in Use

- Operating systems need to be able to handle multiple things at once (MTAO)
  - *processes, interrupts, background system maintenance*
- Servers need to handle MTAO
  - *Multiple connections handled simultaneously in a Web Server*
- Parallel programs need to handle MTAO
  - *To achieve better performance*
- Programs with user interfaces often need to handle MTAO
  - *To achieve user responsiveness while doing computation*
- Network and disk bound programs need to handle MTAO
  - *To hide network/disk latency*

There's a Problem!!!

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, What about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  – *What are the possible values of x?*

Thread A	Thread B
x = 1;	
x = y+1;	
	y = 2;
	y = y*2

x=13

Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, What about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  – *What are the possible values of x?*

Thread A	Thread B
	y = 2;
	y = y*2;
x = 1; x = y+1;	

x=5

Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, What about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  - *What are the possible values of x?*

Thread A	Thread B
	y = 2;
x = 1;	
x = y+1;	
	y= y*2;

x=3

Race Condition is the name!!

- **Race condition:** Output of a concurrent program depends on the order of operations between threads
- Cannot make any assumptions about relative speed of threads (i.e. interleaving is a given)
- Non-determinism is omnipresent:
  - *Scheduler's decision depends on many factors*
  - *Processor architecture (e.g., variable clock rate, out-of-order execution)*

Race Condition for Instruction Set

- Simple threaded code (assume x=0)

Thread1	Thread2
x=x+1;	x=x+1;

Compiler Generated (because it can only use things from the Instruction Set):

load r, x
add r, r, 1
store x, r

Pre-emption

Values of x can be 1 or 2 depending on the order of execution

load r1, x
add r1, r1, 1
store x, r1

load r2, x
add r2, r2, 1
store x, r2

X=2

load r1, x

load r2, x
add r2, r2, 1
store x, r2

add r1, r1, 1
store x, r1

X=1

Race Condition for Out-of-Order Execution

- Most architectures support (I should say "require") this feature
  - *Pipelined processors cannot achieve peek performance without it*
- The idea is simple: compilers may reorder "unrelated" instructions like the following:

```
//global variables
bool done = false;
int data = -1;

Thread1 (){
   // takes a long time
   data = longfunction();
   done = true;
}
```

reorder

```
//global variables
bool done = false;
int data = -1;

Thread1 (){
   done = true;
   // takes a long time
   data = longfunction();
}
```

Out-of-Order Execution causing Race Condition

- The problem arises when there is an inter-thread dependency
- Because of thread 2's wait, the 2 lines in Thread1() are no longer independent
- However, the compiler has no way to tell that!!!

```
// thread 1's function,
// REORDERD
Thread1 (){
   done = true;
   data = longfunction();
}
```

```
// thread 2's function
Thread2 (){
  while (!done); // wait
  int newdata = compute (data);
}
```

Thread API

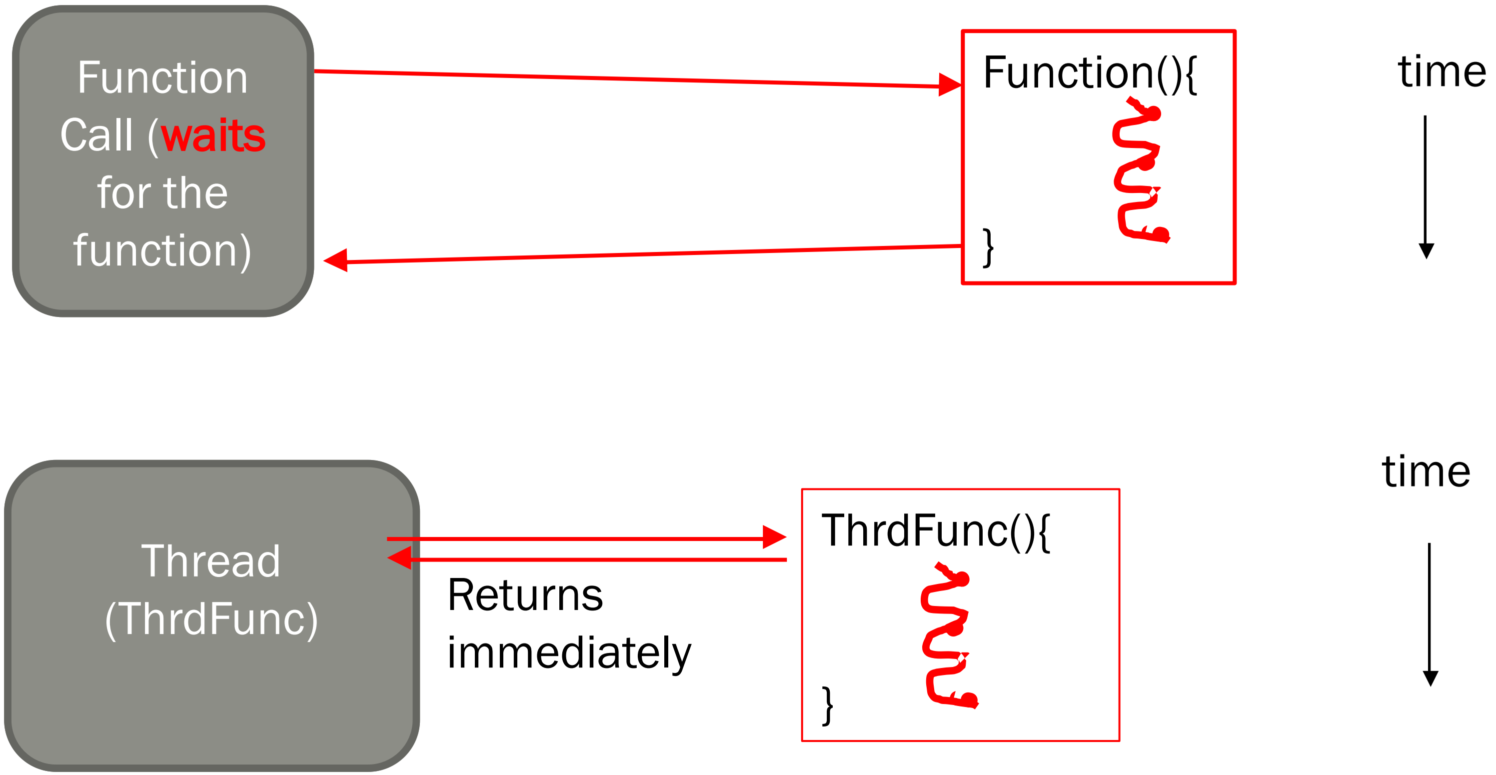
- We will discuss thread API from C++11 which is cross-platform
  - *But for many other things, we will still use Linux for our PAs*
- Here is an example:

```cpp
#include <iostream>
#include <thread>
#include <unistd.h>
using namespace std;

void foo() {
  sleep(3);
  cout<<"foo done"<<endl;
}
void bar(int x){
  sleep(1);
  cout <<"bar done"<<endl;
}
```

```cpp
int main() {
thread foothrd (foo); //calls foo() in a
thread
thread barthrd (bar,0);//calls bar(x) in a
thread

cout<<"main, foo and bar would now execute
concurrently..."<<endl;
// synchronize threads:
foothrd.join();// pauses until foo
finishes
barthrd.join();// pauses until bar
finishes
cout<<"foo and bar completed."<<endl;
return 0;
}
```

Thread Execution

- Creating a thread is very similar to calling a function directly, with a slight difference shown below:
  - *A regular function call is blocking, i.e., caller waits for callee*
  - *thread does NOT*
- So, this is like fork(), creates the thread, calls the function inside, but does not wait



Race Condition Demonstration

- Start 2 (or more) threads that increment a shared variable times
  - *Use pointer to data so that threads share*
- Use a large number for x to ensure adequate overlap
- Notice that the output is different every time

```
void func (int& p, int x) {
	// increment *p x times
	for (int i=0; i<x; i++)
		p = p + 1;
}
int main(int ac, char** av) {
	int data = 0;
	int times = atoi (av [1]);
	// start 2 threads to increment
	thread t1 (func, data, times);
	thread t2 (func, data, times);

	t1.join();
	t2.join();
	cout<<"data="<<data<<endl;
	return 0;
}
```

```
osboxes@osboxes:~/$ ./a.out 10000000
data = 15003189
osboxes@osboxes:~/$ ./a.out 10000000
data = 13380972
osboxes@osboxes:~/$ ./a.out 10000000
data = 12565682
```