Threads 1

Outline

- what are threads and processes
- how to make a thread
- how to synchronize threads
- how to communicate between threads.

Process

What's a process?

The class definition of a process is a program *in execution.* It has

- a virtual address space
- hardware context (registers, stack, file handles, etc.)
- O.S. info about priority, other resources being used

Processes

- Reality: each CPU can only run one program at a time
- Fiction to user: many people getting short (~10-100 ms) time slices
  - pseudo-parallelism → *multiprogramming*
  - modeled as sequential processes
  - *context switch*

Context Switch?

Processes get a fairshare of the CPU. Need to context switch, move one process off and another on the CPU

- What's involved in doing such a switch (do you imagine)

Time and latency

https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/

Maybe 2 υs to switch (4000 clock ticks)

What's the latency here?

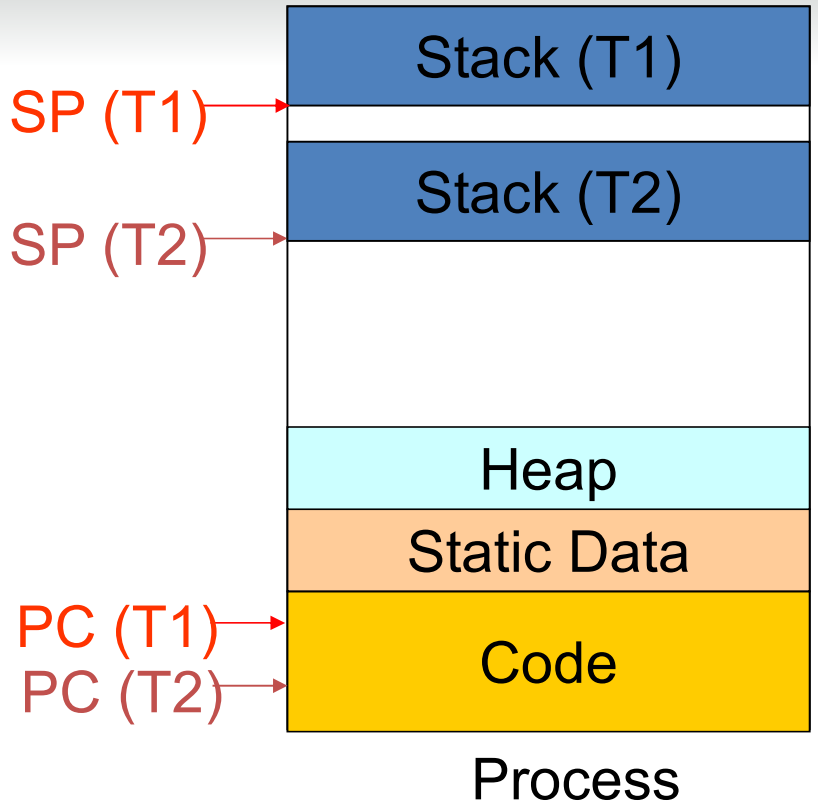
Thread

Threads

- process: address space + single thread of control
- sometimes want multiple threads of control (flow) in same address space
- quasi-parallel
- threads separate resource grouping & execution
- thread: program counter, registers, stack
- also called lightweight processes
- multithreading: avoid blocking when waiting for resources
  - multiple services running in parallel
- state: running, blocked, ready, terminated

What is a Thread?

- Execution context
  - Program counter (PC)
  - Stack pointer (SP)
  - Data registers



Why threads?

- Parallel execution
- Shared resources → faster communication without serialization
- faster to suspend if some are I/O-bound → overlap computation and I/O
- easy porting to multiple CPUs

Thread variants

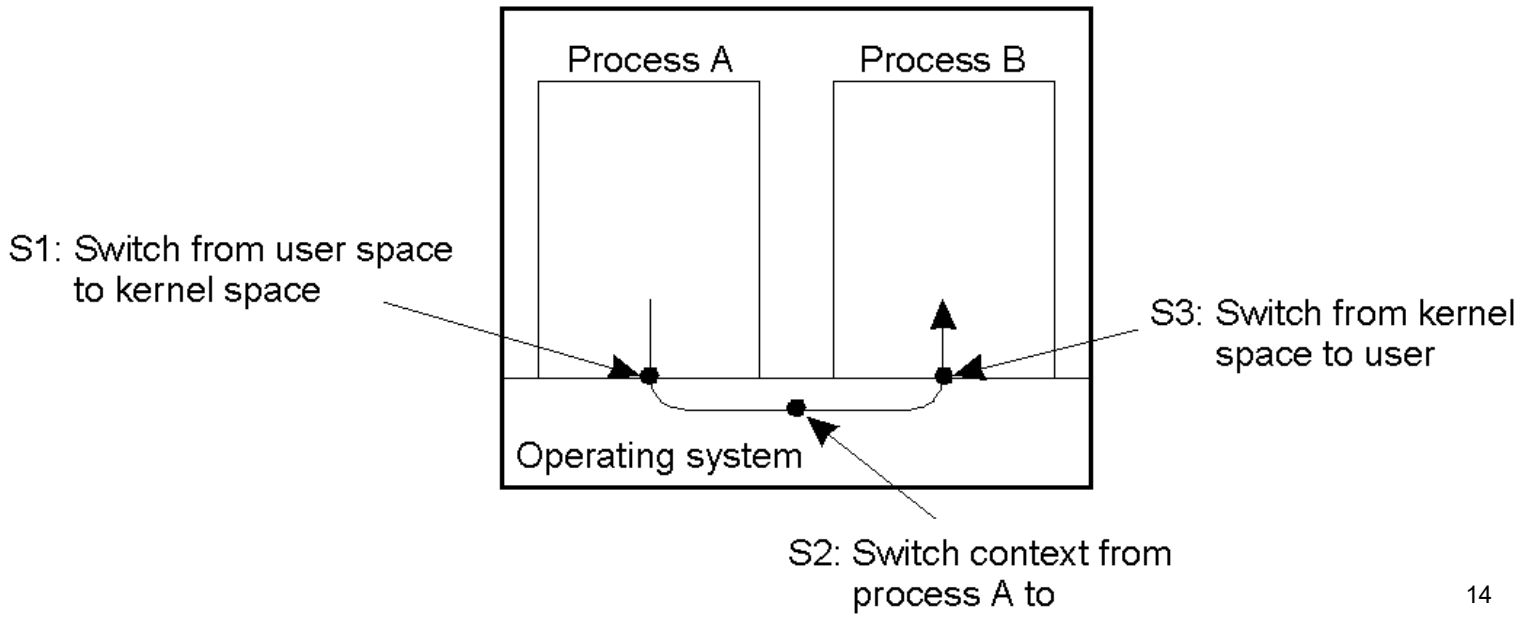
- POSIX (pthreads)
- Java threads
- Windows threads

Process vs. Thread (1)

- Process: unit of allocation
  - Resources, privileges, etc
- Thread: unit of execution
  - PC, SP, registers
- Each process has one or more threads
- Each thread belong to one process

Process vs. Thread (2)

- Processes
  - Inter-process communication is expensive: need to context switch
  - Secure: one process cannot corrupt another process



Process vs. Thread (3)

- Threads
  - Inter-thread communication cheap: can use process memory and may not need to context switch
  - Not secure: a thread can write the memory used by another thread

Benefits of Threads

- Less time to create a new thread than a process (**maybe**, OS dependent)
- Less time to terminate a thread than a process (ditto)
- Switching between threads takes less time that switching processes (yes!)
- Threads can communicate with each other via shared memory
  - without invoking the kernel

Threads

- Several actions that affect all of the threads in a process
  - The OS must manage these at the process level.
- Examples:
  - Suspending a process involves suspending all threads of the process
  - Termination of a process, terminates all threads within the process

Activities similar to Processes

- Threads have execution states and may synchronize with one another.
  - Similar to processes
- We look at these two aspects of thread functionality in turn.
  - Startup/execution
  - Synchronization

Concurrency vs. Parallel Processing

Concurrency means that multiple tasks are being processed at the same time. Not necessarily executing simultaneously

Parallel Processing means that multiple tasks are running at the same time (are executing on the CPU)

Why concurrent?

It is convenient to assign elements of execution to a thread even if that thread is not presently running.

Web browser has multiple threads, one associated with each tab (for example).

Maintains a state of execution!

C++11 threads

two models in C++11

There are really two models in C++11:

- thread model
- async task model

The former is more standardized, the latter more "interesting" from a task view.

built atop a thread implementation

C++11 attempts to be more generic as it provides a model upon which different architectures and be utilized:

- pthreads for Unix
- ConcRT (concurrency runtime) windows

thread

```
#include<thread>

thread(callable object)
```

thread constructor *copies* the callable object to the local thread space and *starts* that separate thread by invoking the callable object

1.1

```
#include<iostream>
using std::cout; using std::endl;
#include<thread>
using std::thread;

void a_fun(){
  cout << "Hello World"<<endl;
}

int main(){
  thread thrd(a_fun);
  thrd.join();
}
```

argument is a callable function

declare the thread obj

either join or detach

compiling

On linux:

```
g++ -pthread file_to_compile.cpp
```

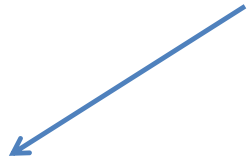
underlying implementation being used to do the C++ threading model

1.2

```
#include<iostream>
using std::cout; using std::endl;
#include<thread>
using std::thread;

int main(){
  thread thrd(
     [](){cout << "Hello World" << endl;}
  );
  thrd.join();
}
```

start thread
on lambda



1.3

```
#include<iostream>
using std::cout; using std::endl;
#include<thread>
using std::thread;
#include<string>
using std::string;

class MyClass{
 private:
  string my_mesg_;
 public:
  MyClass(string s) : my_mesg_(s){};
  void operator()(){
    cout << my_mesg_ << endl;
  }
};

int main(){
  MyClass mc("Hello World");
  thread thrd(mc);
  thrd.join();
}
```

constructor via init list

op() overload makes and instance callable

invoke the instance

What you cannot do

- As shown, you cannot return a value from a thread
  - you can change a reference passed in
  - we'll see other ways, but not this way
- You cannot ever copy a thread
  - if you pass one around, it has to be by reference or by a move.

join vs detach

The thread starts *immediately* and *independently* of the main thread.

The question is how they sync:

- `join`, caller waits at the join call for the thread to end
- `detach`, caller ignores the thread and each proceeds independently
  - often called a *daemon*

when to call

You only have to call join/detach before the thread object/var is destroyed:

- you can join/detach long after the thread has finished
- if you detach, the thread may run long after the thread object is detached and after the main ends

call once and done

If you have detached a thread, you can no longer communicate with it.

- but you can still sync with it (more later)

If you have joined a thread, you cannot join it again

`.joinable()` is a boolean that indicates whether you can join a thread

when does a thread end

- if the associated thread var is destroyed (goes out of scope)
  - at that point the destructor calls terminate, ending the thread.
- if the thread throws an exception
  - thread ends

what happens with exceptions

If the thread throws an exception and it is not handled, then the process halts.

If the main throws and exception before the thread is joined, the thread is destroyed and the thread terminated

exceptions are really tricky

We will see down the line that handling exceptions can be tricky.

The rule is basically "If something can go wrong it will" with concurrency.

We need to be extra careful.

beware detach

- if possible, only work with local/copied values. This avoids a thread trying to access destroyed values in another context
- if using globals, make sure they stay in scope until the thread ends
  - we'll see how to do this later