EC 440 – Introduction to Operating Systems Project 2 – Discussion

Manuel Egele

Department of Electrical & Computer Engineering

Boston University

Preemptive User Mode Threading Library

Preemptive

- CPU is switched independently of the process behavior
  - A clock interrupt is required

User Mode

- No support from kernel necessary
  - Portable (i.e., works even if kernel does not explicitly support threads)
  - If something goes wrong (e.g., crashes) only your program dies not the entire OS

Threads

- … next slide …

Library

- i.e., only the functions required by the project description
- must not have `main` in your library

From Class...

Scheduling Algorithms

Non-preemptive

- CPU is switched when process
  - has finished
  - executes a yield()
  - blocks

Preemptive

- CPU is switched independently of the process behavior
  - A clock interrupt is required

Preemptive User Mode Threading Library

Preemptive

- CPU is switched independently of the process behavior
  - A clock interrupt is required

User Mode

- No support from kernel necessary
  - Portable (i.e., works even if kernel does not explicitly support threads)
  - If something goes wrong (e.g., crashes) only your program dies not the entire OS

Threads

- … next slide …

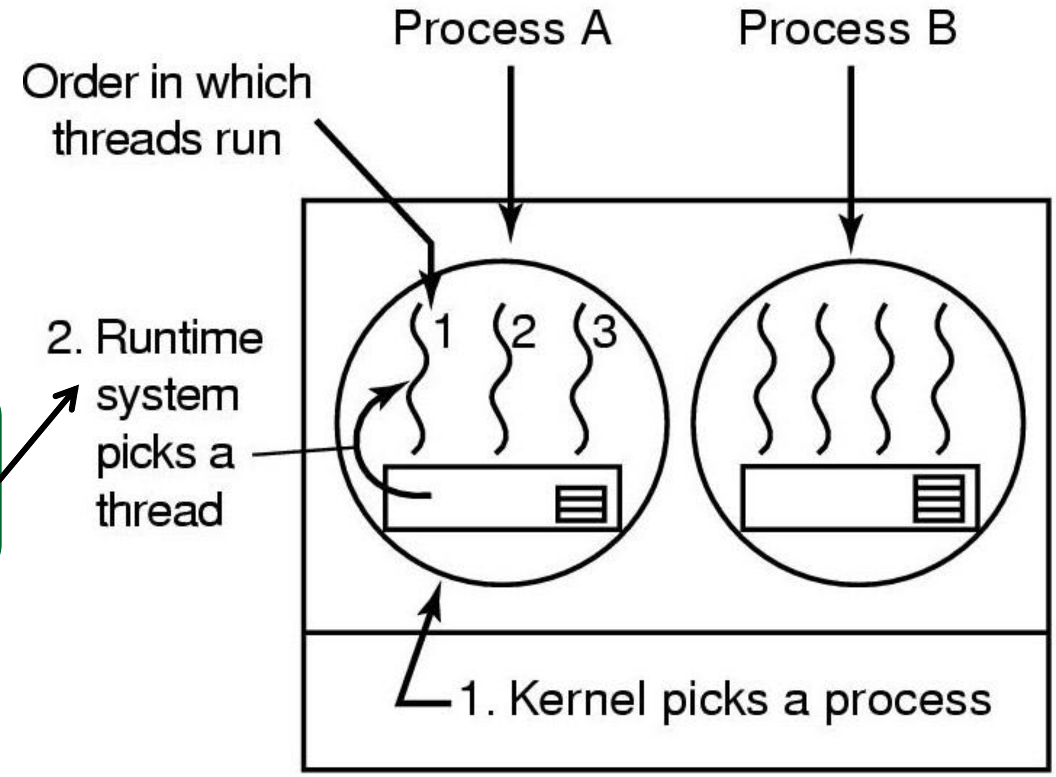
Library

- i.e., only the functions required by the project description
- must not have `main` in your library

Thread Scheduling

From Class...

We will build this!



Process A
Process B
Order in which threads run
1
2
3
2. Runtime system picks a thread
1. Kernel picks a process

Possible: A1, A2, A3, A1, A2, A3

Not possible: A1, B1, A2, B2, A3, B3

Preemptive User Mode Threading Library

Preemptive

- CPU is switched independently of the process behavior
  - A clock interrupt is required

User Mode

- No support from kernel necessary
  - Portable (i.e., works even if kernel does not explicitly support threads)
  - If something goes wrong (e.g., crashes) only your program dies not the entire OS

Threads

- … next slide …

Library

- i.e., only the functions required by the project description
- must not have `main` in your library

Threads

Multiple threads of execution can run in the same *process*

Multiple threads (in the same process) share

- Common address space (shared memory)
- Open files
- Process, user, and group IDs

Cool! But not too important for proj2.

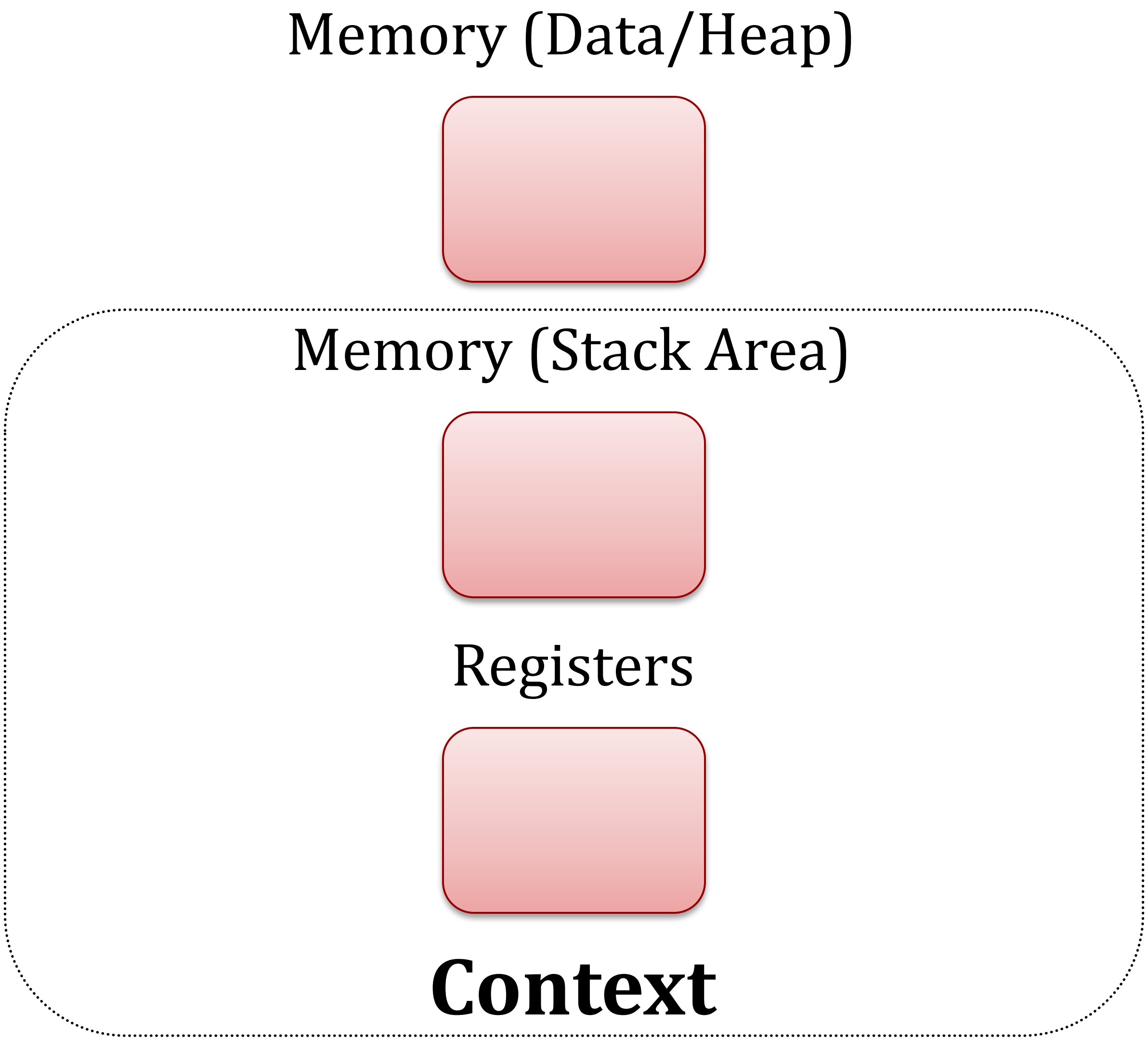
Each thread has its own *context*, consisting of

- Code
- Program counter
- Set of registers
- Stack

Really ***important*** for proj2!

From Class…

Threads (Context)



Memory (Data/Heap)

Memory (Stack Area)

Registers

Context

Implementation Requirements

Implement three pthreads functions:

1. `pthread_create`
2. `pthread_exit`
3. `pthread_self`

Schedule threads

4. Context switch every 50ms in round robin

1.) `phtread_create()`

2.) `phtread_exit()`

What's a good sequence of implementing these 4 components?

a.k.a. Where do I start?

3.) `pthread_self()`

(1 line of code)

4.) `schedule()`

1.) `phtread_create()`

2.) `phtread_exit()`

What does each of these 4 components do?

3.) `pthread_self()`
(Last)

4.) `schedule()`

1) `phthread_create()`

Create a new **thread context** for this thread and set it to READY

2) `phthread_exit()`

Clean up all resources that were allocated for this thread in `pthread_create()`

3) `pthread_self()` (Last)

Still: Where do I start?!

4) `schedule()`

Perform a **context switch** from the current thread to the next thread that’s READY

Possible Development Strategy

Library alone can’t run (i.e., can’t test either)

Incrementally build the threading library and an example program to test the most recently added functionality during development.

- Sounds good, but in what order?
- Is there an inherent sequence in these four components?

Simple Multi-Threaded Program

```
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>

#define THREAD_CNT 3

// waste some time
void *count(void *arg) {
  unsigned long int c = \
    (unsigned long int)arg;
  int i;
  for (i = 0; i < c; i++) {
    if ((i % 1000) == 0) {
      printf("id: %x cntd to %d of %ld\n", \
      (unsigned int)pthread_self(), i, c);
    }
  }
  return arg;
}
```

```
int main(int argc, char **argv) {
  pthread_t threads[THREAD_CNT];
  int i;
  unsigned long int cnt = 10000000;

  //create THRAD_CNT threads
  for(i = 0; i<THREAD_CNT; i++) {
    pthread_create(
      &threads[i], NULL, count,
      (void *)((i+1)*cnt));
  }
  //join all threads ... not important for
  //proj2
  for(i = 0; i<THREAD_CNT; i++) {
    pthread_join(threads[i], NULL);
  }
  return 0;
}
```

Get it on piazza or at:
ec440.bu.edu:/usr/local/ec440/proj2/sample-program/

Execution of a Multi-Threaded Program

Program execution	Implementation Task
1. Program starts	(nothing to do)
2. Launches n threads	`pthread_create()` ①
3. Schedule threads s.t. each thread gets a fair share of CPU time	`schedule()` ②
4. Threads that are complete, exit	`pthread_exit()` ③
5. Program collects results from threads	(nothing to do, for proj2)
6. Program exits	“special” case in `pthread_exit()`

pthread_create()

Create new thread context & mark it READY

- Thread context (and more) is captured in the thread control block (TCB)
- What is a thread's context?
  - Registers
  - Stack
- What else does the TCB need?
  - State (READY, EXITED, RUNNING, etc.)
  - Exit status of the thread (in proj2 it's constant 0)

schedule()

We want to `schedule()` every 50ms

- Set an alarm to go off every 50ms
- In the handler do:
  1. Preserve context of the currently executing thread
  2. Choose the next thread to run (round robin)
  3. Context switch to the new thread (i.e., restore the new thread's context)

When To Schedule

Must schedule

- a thread exits `pthread_exit()`
- a thread blocks (I/O, semaphore, etc.)

May schedule

- new thread is created `pthread_create()`
- I/O interrupt
- clock interrupt `alarm-handler()`

Define Thread Control Block

Data structure to store info about threads

```
struct thread {
        Thread id
        Information about the state of the thread
                (its set of registers)
        Information about its stack
                (a pointer to thread's stack area)
        Information about the status of the thread
                (ready, running or exited)
 };
```

pthread_create()

1. Create new TCB
   - Stack
     - Hint: Draw the stack diagram of the empty stack at pthread_create()
   - Registers, in particular
     - PC – Program Counter
     - SP – Stack Pointer
     - How? Remember `jmp_buf` from `setjmp/longjmp`?
2. Once TCB is initialized set state ← READY
3. Call schedule()

pthread_create()

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

thread ← just the new id

attr ← always NULL, i.e., don't care

start_routine … this is the address of where our thread should start execution (i.e., a pointer to the start_routine function, cf. function pointer)

arg … this is the only argument for the new thread (i.e., start_routine)

pthread_exit()

1. Free all resources for the current thread.
2. Set the thread's state to EXIT
3. Must automatically be called when start_routine finishes (i.e., returns)! How?

pthread_self()

- Return the thread-id of the currently running thread (at any given time there can only be one thread running)
- The scheduler is the only component that can switch threads
- Thus, the scheduler can maintain a global variable that contains the thread-id of the currently running thread.

```
pthread_t pthread_self(void) {
    return gCurrent;
}
```

Preemptive User Mode Threading Library

Preemptive

- CPU is switched independently of the process behavior
  - A clock interrupt is required

User Mode

- No support from kernel necessary
  - Portable (i.e., works even if kernel does not explicitly support threads)
  - If something goes wrong (e.g., crashes) only your program dies not the entire OS

Threads

- … next slide …

Library

- i.e., only the functions required by the project description
- must not have `main` in your library

How to Compile & Test

Remember to

```
#include<pthread.h>
#include “ec440threads.h”
```

in your sources (or copy contents of `ec440threads.h` into your source)

Running `make` must produce a `threads.o` ELF executable. You can get this via

```
$ gcc -Werror -Wall -g -c -o threads.o threads.c
```

Link this with your test file (e.g., main.c)

```
$ gcc -Werror -Wall -g -o main main.c threads.o
```

Exactly what our `makefile` does, provided you have your implementation in `threads.c` and `main.c`

Get it from: ec440.bu.edu:/usr/local/ec440/proj2/makefile or piazza

Things Missing (Incomplete List)

- First time `pthread_create` is called it must:
  - set up all data structures
  - set up the scheduler
  - make a TCB for the main program

Questions?

pthread_create()

`void *(*start_routine)(void *)`

Where does this go?

Hint: The thread must start execution there!

Answer: That's the PC for the *new thread*.

`void *arg`

Where does this go?

Hint: This is an argument to the `start_routine` function!

Answer: In AMD64 calling convention first six arguments are passed in registers (`RDI, RSI`, …). That's where this goes, in `RDI` for the *new thread*.

Read on! …

pthread_create() cont.

While it would be easy to store start_routine in JB_PC (remember ptr_mangle), we cannot easily ensure that arg will be the first argument (in EDI) when start_routine gets “called” (or rather scheduled).

To solve this problem, your implementation of pthread_create should store start_routine in R12 and arg in R13 and ensure that the new thread commences as start_thunk

start_thunk moves the value from R13 to RDI and “jumps” to R12, hence faking a call to start_routine

Use ec440threads.h from piazza or bandit

schedule()

We want to `schedule()` every 50ms

- Set an alarm to go off every 50ms
- In the handler do:
  1. Preserve context of the currently executing thread
     How? Call `setjmp()` & preserve the `jmp_buf`.
     Where?
  2. Choose the next thread **T** to run (round robin)
  3. Context switch to **T** (i.e., restore **T**'s context)
     How? Get `jmp_buf` for **T** and call `longjmp()` with it.