10/14 Threads Execution

Threads

- → TCB
- → Within the same process, heap, code, data is shared, stack is private
- → Kernel threads
  - → can execute in user mode, the main thread in every single threaded process
  - → each process has a kstack (interrupt stack)
    - ↳ each thread has its own kstack

```
int main() {
    test();
    foo();
    bar();
}
```

single threaded

```
int main() {
    thread_create(foo);
    thread_create(bar);
    thread_join() ×2;
}
```

foo()

bar();

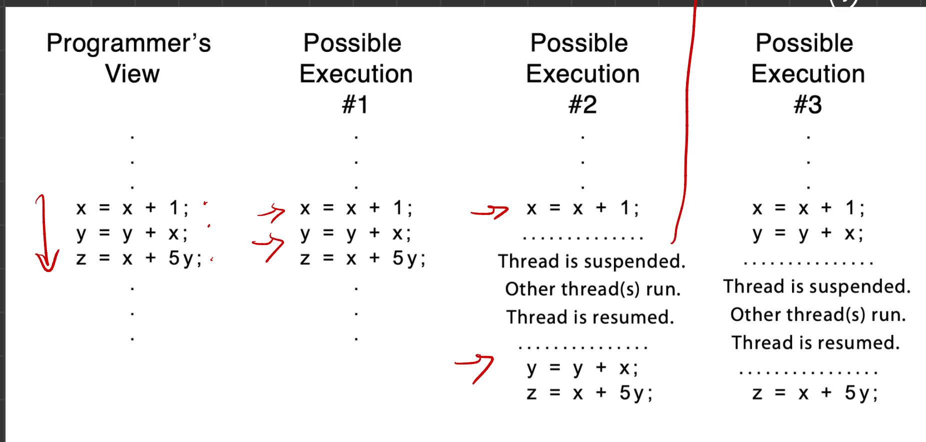
multi-threaded.

Thread Abstraction

→ an abstraction for dedicated CPU

Why might a thread suspend?

1. timer interrupts. (fair sharing).
2. I/O.
3. thread error out (terminates)
4. thread_yield (give up voluntarily)



★ need to support pause & resume.

Context Switch

```
// Give up the CPU for one scheduling round.
void yield(void) {
  acquire(&ptable.lock); // DOC: yieldlock
  myproc()->state = RUNNABLE;   = READY
  sched();
  release(&ptable.lock);
}
```

```
void sched(void) {
  int intena;

  if (!holding(&ptable.lock))
    panic("sched ptable.lock");
  if (mycpu()->ncli != 1) {
    cprintf("pid : %d\n", myproc()->pid);
    cprintf("ncli : %d\n", mycpu()->ncli);
    cprintf("intena : %d\n", mycpu()->intena);

    panic("sched locks");
  }
  if (myproc()->state == RUNNING)
    panic("sched running");
  if (readeflags() & FLAGS_IF)
    panic("sched interruptible");

  intena = mycpu()->intena;
  swtch(&myproc()->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

```c
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void scheduler(void) {
  struct proc *p;

  for (;;) {
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if (p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      mycpu()->proc = p;
      vspaceinstall(p);
      p->state = RUNNING;
      swtch(&mycpu()->scheduler, p->context);
      vspaceinstallkern();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      mycpu()->proc = 0;
    }
    release(&ptable.lock);
  }
}
```

process table

installs VAS.

running

switches to the next proc.

```
# Context switch code. We only need to save callee save registers
# as all other registers are saved before we reach here.
# arg0 -- context (in struct process) for the old process
# arg1 -- context (in struct process) for the new process
#
# note that xk only switches to/from the scheduler process
# the scheduler process picks the next thread to run
#
.globl swtch
swtch:
  push %rbp
  push %rbx
  push %r11
  push %r12
  push %r13
  push %r14
  push %r15

  mov %rsp, (%rdi)
  mov %rsi, %rsp

  pop %r15
  pop %r14
  pop %r13
  pop %r12
  pop %r11
  pop %rbx
  pop %rbp

  ret

```

saving old process registers

old process state

sets the new process state

popping new process's registers

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

int global_x = 0;

void* increment() {
    global_x += 1;
    return NULL;
}

int main(int argc, char** argv) {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, increment, NULL);
    pthread_create(&tid2, NULL, increment, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("%d\n", global_x); // minimum? maximum?

    return 0;
}
```

tid1 tid2

main

```
(gdb) disas /m increment
Dump of assembler code for function increment:
6       void* increment() {
   0x0000000000401146 <+0>:     push   %rbp
   0x0000000000401147 <+1>:     mov    %rsp,%rbp

7           global_x += 1;
   0x000000000040114a <+4>:     mov    0x2ee8(%rip),%eax        # 0x404038 <global_x>
   0x0000000000401150 <+10>:    add    $0x1,%eax
   0x0000000000401153 <+13>:    mov    %eax,0x2edf(%rip)        # 0x404038 <global_x>

8           return NULL;
   0x0000000000401159 <+19>:    mov    $0x0,%eax

9       }
   0x000000000040115e <+24>:    pop    %rbp
   0x000000000040115f <+25>:    ret
```

Actually 3 instructions.

Possible outcomes

1) - 2 = t1 t2

execute one after the other.

2) - 1

t1 t2

read global_x (0) (into a register)

x = 1

increment — increment the register value

write global_x (1) — writes the register value

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

int global_x = 0;
pthread_t tids[100];

void* increment() {
  global_x += 1;
  return NULL;
}

int main(int argc, char** argv) {
  for (int i=0; i<100; i++) {
    pthread_create(&tids[i], NULL, increment, NULL);
  }
  printf("%d\n", global_x); // minimum? maximum?

  for (int i=0; i<100; i++) {
    pthread_join(tids[i], NULL);
  }
  printf("%d\n", global_x); // minimum? maximum?

  return 0;
}
```

read x to reg
add 1 to reg
write reg to x

0: why?

100: all execute in turn

→ 100

1: t1
read x=0
add 1 to reg
write reg to x
(1)

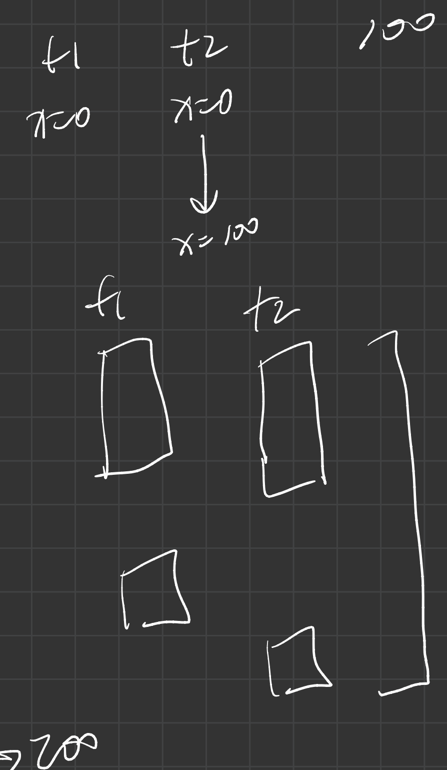
t2 – t100
all execute in order

x=99

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

int global_x = 0;
pthread_t tids[2];

void* increment100() {
  for (int i=0; i<100; i++) {
    global_x += 1;
  }
  return NULL;
}

int main(int argc, char** argv) {
  for (int i=0; i<2; i++) {
    pthread_create(&tids[i], NULL, increment100, NULL);
  }

  for (int i=0; i<2; i++) {
    pthread_join(tids[i], NULL);
  }

  printf("%d\n", global_x); // minimum? maximum?

  return 0;
}
```

read x
add 1
write x

Smaller than 100, so what is it?



Problem: Unpredictable Output Based On Different Executions

→ multiple threads access a shared variable w/out protections (data race)

→ explicit synchronization primitives: locks