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;
  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 (termination).
4. Thread yield (give up voluntarily).

Need to support pause & resume.

<table>
<thead>
<tr>
<th>Programmer’s View</th>
<th>Possible Execution</th>
<th>Possible Execution</th>
<th>Possible Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x = x + 1; y = y + x; z = x + 5y;</td>
<td>x = x + 1; y = y + x; z = x + 5y;</td>
<td>x = x + 1; y = y + x; z = x + 5y;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x = x + 1; y = y + x; z = x + 5y;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thread is suspended. Other thread(s) run. Thread is resumed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>y = y + x; z = x + 5y;</td>
<td></td>
</tr>
</tbody>
</table>
209 // Give up the CPU for one scheduling round.
210 void yield(void) {
211     acquire(&ptable.lock); // DOC: yieldlock
212     myproc()->state = RUNNABLE; = READY.
213     sched();
214     release(&ptable.lock);

187 void sched(void) {
188     int intena;
189
190     if (!holding(&ptable.lock))
191         panic("sched ptable.lock");
192     if (mycpu()->ncli != 1) {
193         cprintf("pid : %d\n", myproc()->pid);
194         cprintf("ncli : %d\n", mycpu()->ncli);
195         cprintf("intena : %d\n", mycpu()->intena);
196
197         panic("sched locks");
198     }
199     if (myproc()->state == RUNNING)
200         panic("sched running");
201     if (readeflags() & FLAGS_IF)
202         panic("sched interruptible");
203
204     intena = mycpu()->intena;
205     swtch(&myproc()->context, mycpu()->scheduler);
206     mycpu()->intena = intena;
207 }
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 != RUNNING)
                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);
}
# 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
```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;
}
```

Possible outcomes:
1. $2 = t_1 + t_2$
2. Increment `global_x` (0)
3. Read `global_x` (0)
4. Write the register value
5. Increment the register value

Actually 3 instructions.
```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;
}
```
```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;
}
```
Problem: Unpredictable Output Based On Different Executions

-> multiple threads access a shared variable w/out protections (data race)
-> explicit synchronization primitives - locks