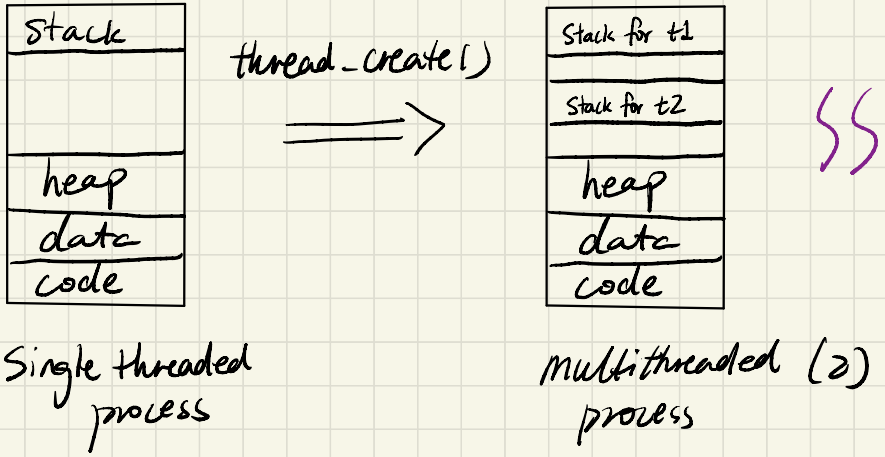
10/11/23 Threads

→ unit of execution (execution state) — PC, SP, regs

→ process = bundle of resources

address space, OS resources

* each has its own user stack & kernel stack



Single threaded process

Multithreaded (2) process

→ Managed & scheduled by the kernel

→ Thread Control Block (TCB)

```
// Per-process state
struct proc {
  struct vspace vspace;          // Virtual address space descriptor
  char* kstack;                  // Kernel stack
  enum procstate state;          // Process state
  int pid;                       // Process ID
  struct proc *parent;           // Parent process
  struct trap_frame *tf;         // Trap frame for current syscall
  struct context *context;       // swtch() here to run process
  void *chan;                    // If non-zero, sleeping on chan
  int killed;                    // If non-zero, have been killed
  char name[16];                 // Process name (debugging)
};
```

thread state

tid

open files (fd)

context switch

managed by the scheduler to pause & resume a thread.

Save current thread's context onto its kernel stack

Switch to the next thread's kernel stack & pop the saved context

if the next thread is from a different process, load new address space & flush the TLB

xk: current thread → scheduler → next thread

(pick new thread to run)

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if (myproc() && myproc()->state == RUNNING &&
    tf->trapno == TRAP_IRQ0 + IRQ_TIMER)
  yield();
```

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

Ready state

```
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;
}
```

Context switch to 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;
    }
```

look for process to switch to

Switch to next process

```
.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
```

swtch.S

→ only need to save callee saved regs.

Pthreads API

PC

-> pthread_create(thread_func, args)

-> pthread_join(tid) wait for tid to exit, any thread can join another

-> pthread_exit(exit_status) terminate the calling thread.

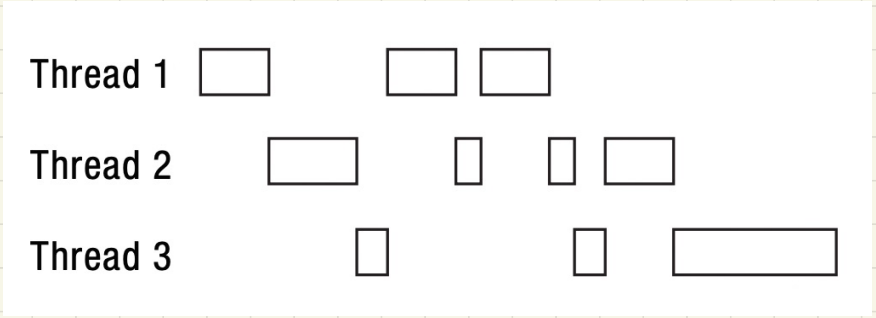
-> pthread_detach upon exit, clean up resources (stack) automatically.
(does not require join)

Threads Execution

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	.....	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended.	.....
.	.	Other thread(s) run.	Thread is suspended.
.	.	Thread is resumed.	Other thread(s) run.
.	.	.....	Thread is resumed.
		y = y + x;	.....
		z = x + 5y;	z = x + 5y;

value of x may change

timer Interrupt



int x = 0; // global var.

data race

t1

{ if(x<1) {

x++;

} // x=2

t2

{ if(x<1) {

x++;

} // x=1

```c
int global_x = 0;

void* thread_func() {
  global_x++;
  return NULL;
}

int main() {
  pthread_t tid1, tid2;

  pthread_create(&tid1, NULL,
  pthread_create(&tid2, NULL,

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

  printf("global_x: %d\n", global_x);

  return 0;
}
```

* not atomic!

```
(gdb) disas /m thread_func
Dump of assembler code for function thread_func:
6       void* thread_func() {
   0x00000000000011a9 <+0>:     endbr64
   0x00000000000011ad <+4>:     push   %rbp
   0x00000000000011ae <+5>:     mov    %rsp,%rbp

7         global_x++;
   0x00000000000011b1 <+8>:     mov    0x2e5d(%rip),%eax        # 0x4014 <global_x>
   0x00000000000011b7 <+14>:    add    $0x1,%eax
   0x00000000000011ba <+17>:    mov    %eax,0x2e54(%rip)        # 0x4014 <global_x>

8         return NULL;
   0x00000000000011c0 <+23>:    mov    $0x0,%eax

9       }
   0x00000000000011c5 <+28>:    pop    %rbp
   0x00000000000011c6 <+29>:    ret
```

global_x might be 1 or 2

3 instr: read x into reg
add 1 to reg
write reg to x

t1

reads x into reg (0)

add 1 to reg (1)

write reg to x (x=1)

t2

reads x into reg (0)

add 1 to reg (1)

write reg to x (x=1)

```c
int global_x = 0;
pthread_t tids[100];

void* thread_func() {
  global_x++;
  return NULL;
}

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

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

  return 0;
}
```

```c
int global_x = 0;

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

int main() {
  pthread_t tid1, tid2;

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

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

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

  return 0;
}
```