# threads

thread = unit of CPU scheduling

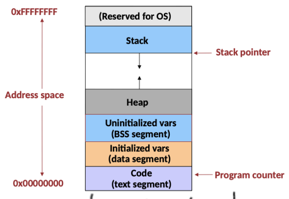
↳ each process has one or more threads within it

same — same — diff
process = address space + OS resources + CPU state

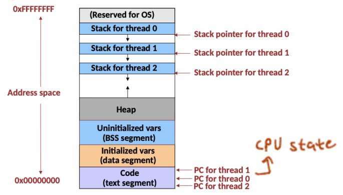
↳ each thread has its own stack, CPU registers, program counter (CPU execution state)

↳ all threads within a same process share the same address space, OS resources

## address space



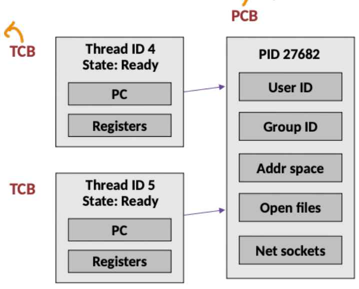
↳ single thread



↳ multiple threads

thread control block

- processor state
- pointer to PCB



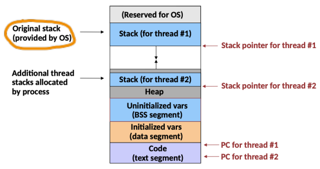
process control block

- address space
- OS resources

no processor state in it

context switching = copy TCB to PCB from ready queue, no need to change address space (switching CPU state)

user level thread address space = you allocate space for stacks in heap



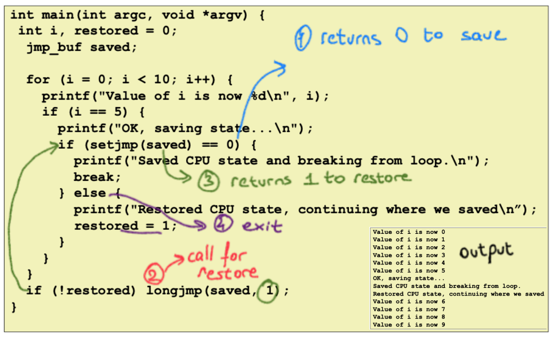
int setjmp( jmp_buf env) = save cpu state

↳ 0 if saving the current state

↳ returns rtrnval if call is from longjmp

void longjmp ( jmp_buf env, int rtrnval) = restore

struct jmp_buf { } → CPU specific fields



```
int main(int argc, void *argv) {
 int i, restored = 0;
  jmp_buf saved;

  for (i = 0; i < 10; i++) {
    printf("Value of i is now %d\n", i);
    if (i == 5) {
      printf("OK, saving state...\n");
      if (setjmp(saved) == 0) {
        printf("Saved CPU state and breaking from loop.\n");
        break;
      } else {
        printf("Restored CPU state, continuing where we saved\n");
        restored = 1;
      }
    }
  }
  if (!restored) longjmp(saved, 1);
}
```

① returns 0 to save

② call for restore

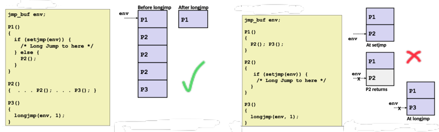
③ returns 1 to restore

④ exit

output

```
Value of i is now 0
Value of i is now 1
Value of i is now 2
Value of i is now 3
Value of i is now 4
Value of i is now 5
OK, saving state...
Saved CPU state and breaking from loop.
Restored CPU state, continuing where we saved
Value of i is now 6
Value of i is now 7
Value of i is now 8
Value of i is now 9
```

- can only jump to func that has not been completed yet →



```
jmp_buf env;

P1()
{
  if (setjmp(env)) {
    /* Long Jump to here */
  } else {
    P2();
  }
}

P2()
{  . . . P2(); . . . P3(); }

P3()
{
  longjmp(env, 1);
}
```

```
jmp_buf env;

P1()
{
  P2(); P3();
}

P2()
{
   if (setjmp(env)) {
    /* Long Jump to here */
   }
}

P3()
{
  longjmp(env, 1);
}
```

preemption = forcefully taking CPU from a thread, and give it to someone else

↳ nonpreemtive threads = CPU allocates the process till it terminates, can be infinitely long

↳ thus threads must call back periodically

↳ yield() = thread voluntarily gives up on CPU - yol ver → context switch

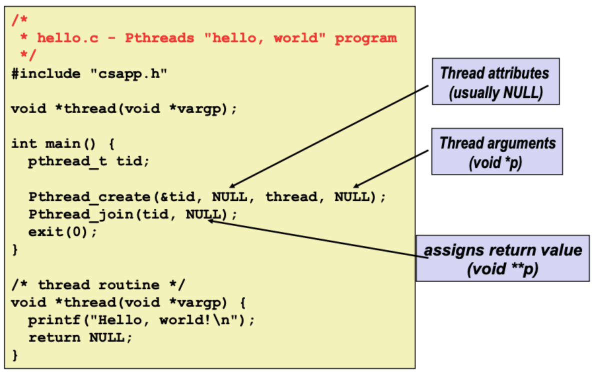
↳ preemtive threads = thread library tells OS to send it a signal periodically (timer)

↳ signal → signal handler → thread library → context switch

kernel level threads = OS can assign priorities to threads, but requires system calls, slower

- in user level threads if one thread blocks, entire process stops/blocked, can't use multiple CPU

posix threads (pthreads) interface =

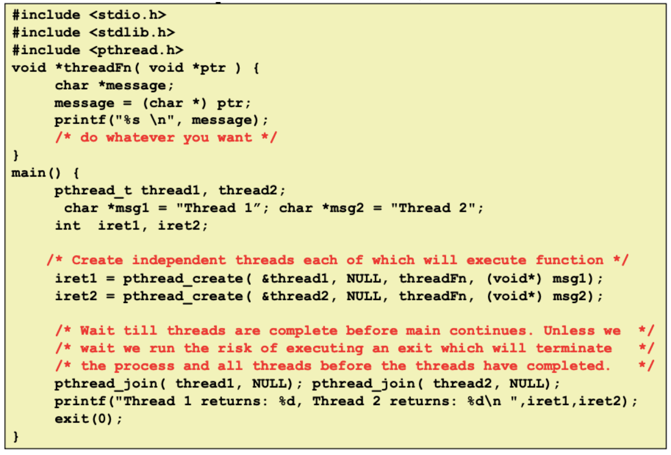


```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *thread(void *vargp);

int main() {
  pthread_t tid;

  Pthread_create(&tid, NULL, thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}

/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
```

Thread attributes (usually NULL)

Thread arguments (void *p)

assigns return value (void **p)



```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *threadFn( void *ptr ) {
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
     /* do whatever you want */
}
main() {
     pthread_t thread1, thread2;
      char *msg1 = "Thread 1"; char *msg2 = "Thread 2";
     int  iret1, iret2;

    /* Create independent threads each of which will execute function */
     iret1 = pthread_create( &thread1, NULL, threadFn, (void*) msg1);
     iret2 = pthread_create( &thread2, NULL, threadFn, (void*) msg2);

     /* Wait till threads are complete before main continues. Unless we  */
     /* wait we run the risk of executing an exit which will terminate   */
     /* the process and all threads before the threads have completed.   */
     pthread_join( thread1, NULL); pthread_join( thread2, NULL);
     printf("Thread 1 returns: %d, Thread 2 returns: %d\n ",iret1,iret2);
     exit(0);
}
```

joinable threads: can be reaped and killed by other threads • must be reaped to free memory resources

detached threads: cannot be reaped or killed by other threads • resources are automatically reaped on termination

singnals to threads = when a process receives a signal, all threads share it

- every thread has its own signal handling → they can react diffently to signal
- but a signal is received only once, random thread will take it