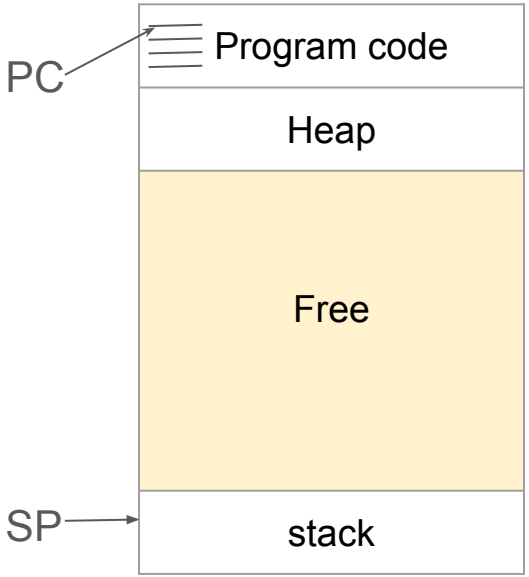
the pthreads API

CS744 Design and Engineering of Computing Systems

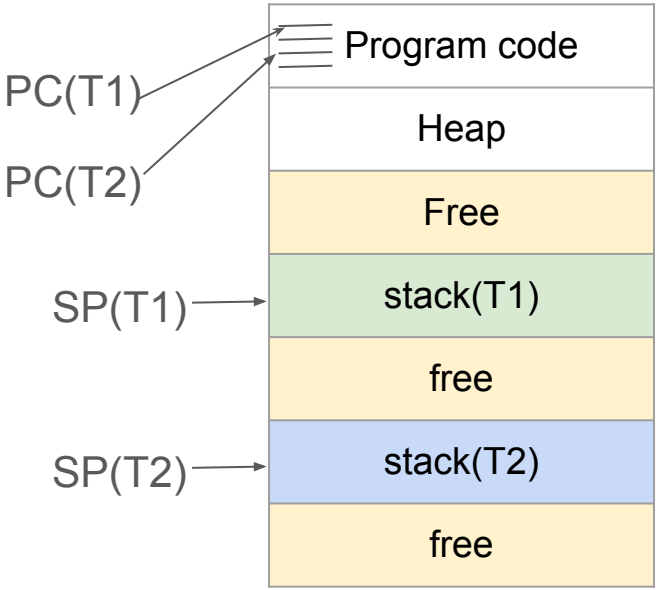
Autumn 2024

Threads

- An independent ordering of instructions
- Another copy of a process that executes independently
- Share same address space (user-level threads)
- Independent address spaces but map to same physical addresses (kernel threads)
- Different PC and stack (either on same process or across processes)



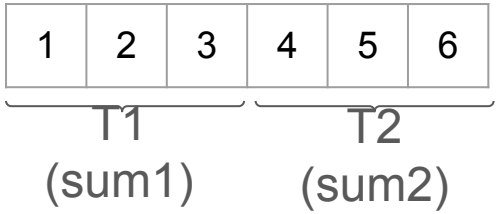
Single threaded



multi threaded
(2 threads)

Why threads?

Example:



Sum = sum1+sum2

Parallelism

Efficient

pthreads

Thread API

To create and control threads

- To use pthread API, include pthread.h header in source code
- To compile, use -pthread flag:

```
gcc -o prog prog.c -pthread
```

Thread creation

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void*),
                   void *arg);
```

thread: pointer to structure of type pthread_t

attr: used to specify any attribute this thread might have.

start_routine: the function this thread start running in

Arg: argument to be passed to the function

Thread creation

Task 1: program to create threads

Why there is no output ?

Main() thread is exiting before other threads finishes

Thread Completion

int **pthread_join**(pthread_t thread, void **value_ptr);

wait for completion of thread

thread: to specify which thread to wait for

**value_ptr: Pointer to the return value you expect to get back

Pass NULL, if we don’t care about the return value

Task 2: Revisit Task 1 with pthread_join

Example: threads with shared data

```
static volatile int counter = 0;
void *mythread(void *arg)
{
   printf("%s\n", (char *) arg);
   int i ;
   for(i=0; i<1000000; i++)
      {
           counter = counter+1;
      }
   return NULL;
}
```

Shared variable

Task 3: threads with shared variable

Example: threads with shared data

Expected output: 2000000

Actual output: Smaller than 2000000

Why ?

Race condition
(simultaneous access to critical section)

Race condition

CS: counter = counter+1

Assemble code:

```
mov eax, [counter]   ; Load the value of 'counter' from memory into EAX
inc eax              ; Increment the value in EAX by 1
mov [counter], eax   ; Store the updated value back to 'counter'
```

Race condition

OS	Thread 1	Thread 2	eax	counter
	Before … mov eax,[counter] inc eax		**0** 20 21	**20** 20 20

Race condition

OS	Thread 1	Thread 2	eax	counter
	Before … mov eax,[counter] inc eax		**0** 20 21	**20** 20 20
Interrupt Save T1 state and restore T2 state			**21** **0**	**20** **20**

Race condition

OS	Thread 1	Thread 2	eax	counter
	Before …		0	20
	mov eax,[counter]		20	20
	inc eax		21	20
Interrupt Save T1 state and restore T2 state			**21** **0**	**20** **20**
		mov eax,[counter]	20	20
		inc eax	21	20
		mov [counter], eax	21	21

Race condition

OS	Thread 1	Thread 2	eax	counter
	Before …		0	20
	mov eax,[counter]		20	20
	inc eax		21	20
Interrupt Save T1 state and restore T2 state			0	20
		mov eax,[counter]	20	20
		inc eax	21	20
		mov [counter], eax	21	21
Interrupt Save T2 state and restore T1 state			**21**	**21**
			21	**20**

Race condition

OS	Thread 1	Thread 2	eax	counter
	Before …		0	20
	mov eax,[counter]		20	20
	inc eax		21	20
Interrupt Save T1 state and restore T2 state			0	20
		mov eax,[counter]	20	20
		inc eax	21	20
		mov [counter], eax	21	21
Interrupt Save T2 state and restore T1 state			**21**	**21**
			21	**20**
	mov [counter], eax		21	21

Race condition

OS	Thread 1	Thread 2	eax	counter
	Before …		0	20
	mov eax,[counter]		20	20
	inc eax		21	20
Interrupt Save T1 state and restore T2 state			0	20
		mov eax,[counter]	20	20
		inc eax	21	20
		mov [counter], eax	21	21
Interrupt Save T2 state and restore T1 state			**21**	**21**
			21	**20**
	mov [counter], eax		21	21 ❌ 22 ✅

Race condition

Concurrent execution may lead to race condition

Solution: Mutual exclusion of critical section and
Atomicity of critical section

How to achieve it: Using locks

Locks

Provide mutual exclusion to a critical section

int **pthread_mutex_lock**(pthread_mutex_t *mutex);

int **pthread_mutex_unlock**(pthread_mutex_t *mutex);

code:

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
// critical section
pthread_mutex_unlock(&lock);
```

Locks

Initialize lock:
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Another way to initialize:
int rc = pthread_mutex_init(&lock, NULL);

To destroy:
pthread_mutex_destroy(&lock);

Task 4: Revisit task 3 with locks

Condition Variables

- Useful when some kind of signaling must take place between threads
- A condition variable (CV) is a queue that a thread can put itself into when waiting on some condition

int **pthread_cond_wait**(pthread_cond_t *cond, pthread_mutex_t *mutex);

- puts the calling thread to sleep, and waits for some other thread to signal it
- The responsibility of wait() is to release the lock and put the calling thread to sleep (atomically); when the thread wakes up (after some other thread has signaled it), it must re-acquire the lock before returning to the caller.

int **pthread_cond_signal**(pthread_cond_t *cond);

- unblock at least one thread blocked on the conditional variable
- Signal wakes up one thread, signal broadcast wakes up all waiting thread

Condition Variables

- A thread calling wait routine:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;
pthread_mutex_lock(&lock);
while (initialized == 0)
    pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

- A thread calling signal routine:

```
pthread_mutex_lock(&lock);
initialized = 1;
pthread_cond_signal(&init);
pthread_mutex_unlock(&lock);
```

Task 5: Conditional variable

Condition Variables

```
while(initialized == 0); // spin
```

This will waste CPU cycle

Why lock is important?

- A thread calling wait routine:

```
while (initialized == 0)
    pthread_cond_wait(&init, &lock);
```

Before calling wait context switch happens

- A thread calling signal routine:

```
initialized = 1;
pthread_cond_signal(&init);
```

More About locks

Locks

How to build a lock?

Objective:

1. Mutual exclusion
2. Fairness
3. Performance

Locks: controlling interrupts

```
void lock() {
        DisableInterrupts();
}
void unlock() {
        EnableInterrupts();
}
```

- This technique is used to implement locks on single processor systems inside the OS
- Disabling interrupts is a privileged instruction and program can misuse it (e.g., run forever)
- Will not work on multiprocessor systems, since another thread on another core can enter critical section

Locks: A simple flag

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
mutex->flag = 0;
}
void lock(lock_t *mutex) {
while (mutex->flag == 1) ;
mutex->flag = 1;
}
void unlock(lock_t *mutex) {
mutex->flag = 0;
}
```

Locks: A simple flag

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
mutex->flag = 0;
}
void lock(lock_t *mutex) {
while (mutex->flag == 1) ;
mutex->flag = 1;
}
void unlock(lock_t *mutex) {
mutex->flag = 0;
}
```

Locks: A simple flag

Thread 1

call lock()

while (flag == 1)

Before setting flag

interrupt: switch to Thread 2

Thread 2

call lock()

while (flag == 1)

flag = 1;

interrupt: switch to Thread 1

flag = 1;

Not providing Mutual exclusion

Hardware Atomic Instructions

Continue in next class….