CS330: Operating Systems

Threads

What is a thread?

- Threads are (almost!) independent execution entities of a single process
- Threads of a single process can be scheduled different CPUs in a concurrent manner. Therefore,
  - Each thread has a different register state and stack
  - At a given point of time, PC of different threads can be different
- How threads are different from processes?
  - Threads of a single process share the address space
  - Context switch between two threads of a process does not require switching the address space

Leverage multi-core systems

- Threads share the address space
  - Global variables can be accessed from thread functions
  - Dynamically allocated memory can be passed as thread arguments

Leverage multi-core systems

- Threads share the address space
  - Global variables can be accessed from thread functions
  - Dynamically allocated memory can be passed as thread arguments
- Example parallel computation models
  - Data parallel processing: Data is partitioned into disjoint sets and assigned to different threads
  - Task parallel processing: Each thread performs a different computation on the same data

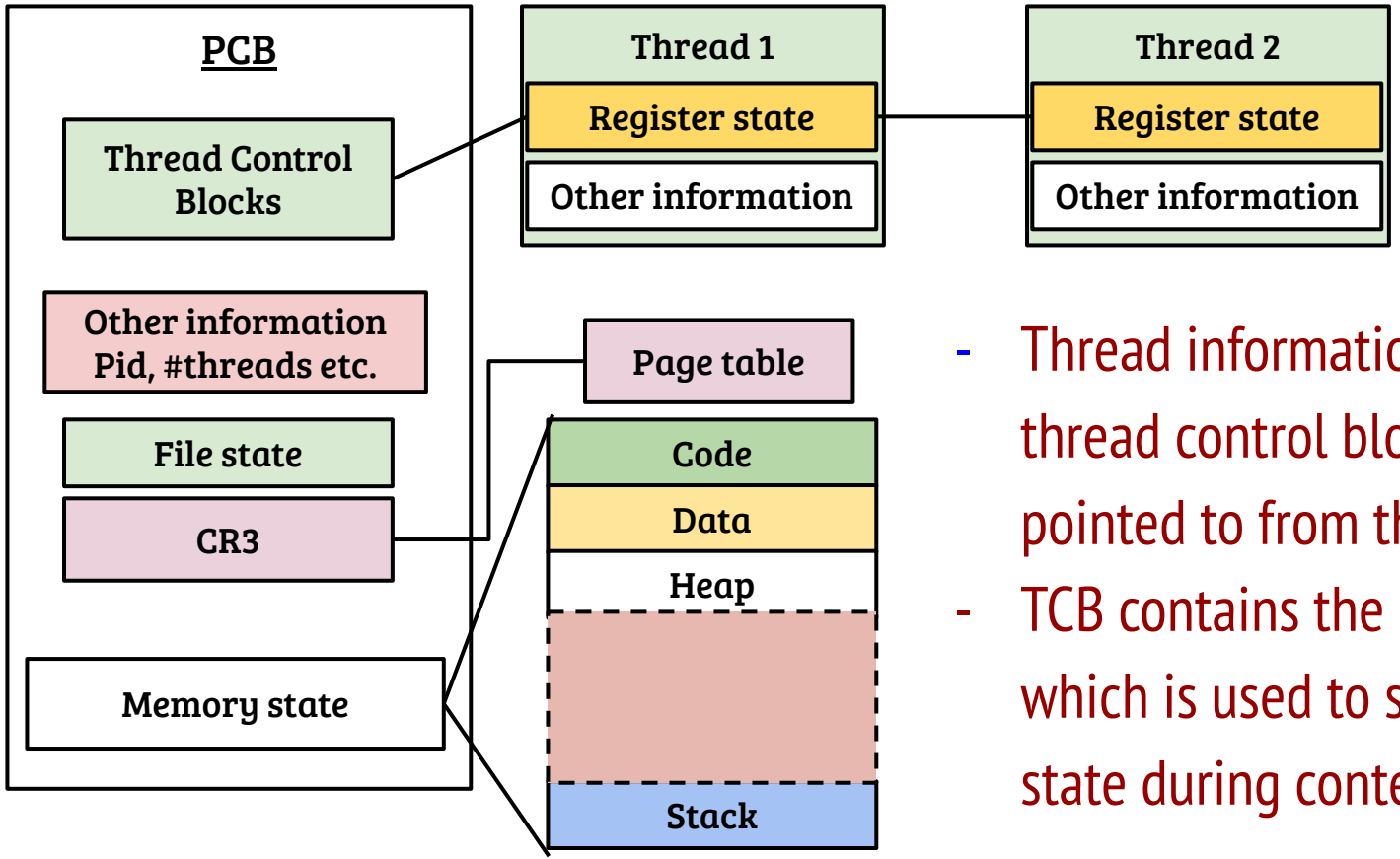
Example: Finding MAX

- Given *N* elements and a function *f*, we are required to find the element *e* such that *f(e)* is maximum
- If the computation time for function *f* is significant, we can employ multithreading with K threads using the following strategy
- Partition N elements into K non-overlapping sets and assign each thread to compute the MAX within its own set
- When all threads complete, we find out the global maximum

Multi-threaded processes

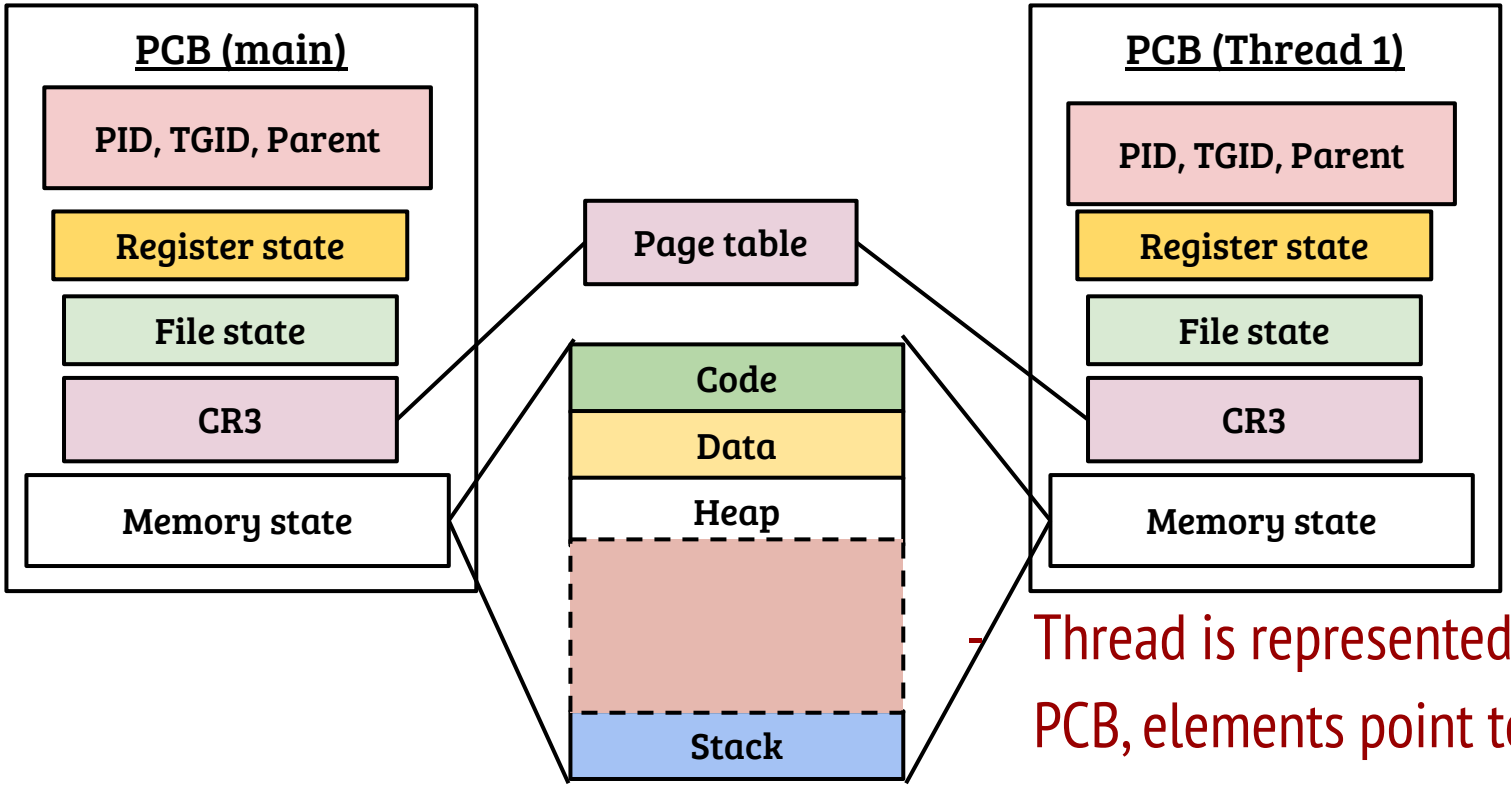
- Why multithreading is useful?
- Efficient execution on multicore systems, overlapping I/O and processing
- How does OS maintain thread related information?
- How stacks for multiple threads are managed?
- What is POSIX thread API? How is it used?

PCB of a multithreaded process



- Thread information is stored in thread control blocks (TCB) which is pointed to from the PCB
- TCB contains the register state which is used to save/restore CPU state during context switch

PCB of a multithreaded process (Linux)

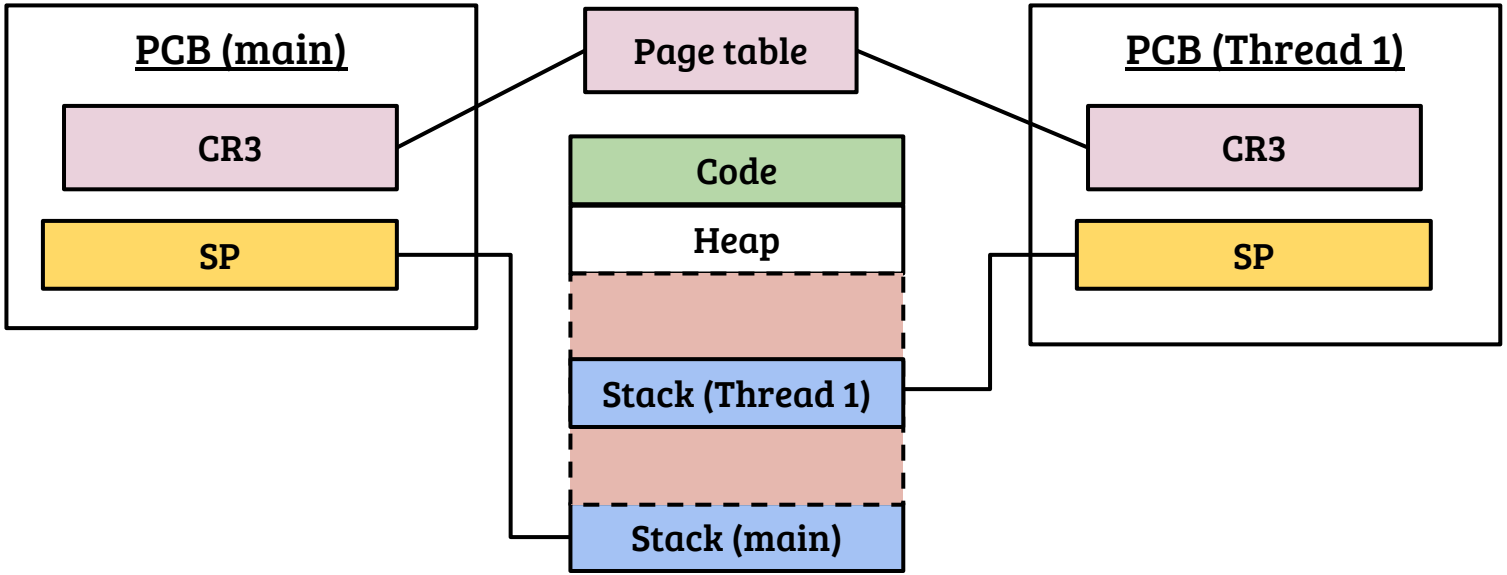


- Thread is represented by a separate PCB, elements point to the structure containing subsystem level info.

Multi-threaded processes

- Why multithreading is useful?
- Efficient execution on multicore systems, overlapping I/O and processing
- How does OS maintain thread related information?
- Maintain thread information using separate PCB or using TCB
- How stacks for multiple threads are managed?
- What is POSIX thread API? How is it used?

Stack for multi-threaded processes



- Stack for threads dynamically allocated from the address space using mmap( ) system call and passed to the OS during thread creation

Multi-threaded processes

Threads are (almost!) independent execution entities of a single process

- Why multithreading is useful?
- Efficient execution on multicore systems, overlapping I/O and processing
- How does OS maintain thread related information?
- Maintain thread information using separate PCB or using TCB
- How stacks for multiple threads are managed?
- Stacks for threads are allocated using memory allocation APIs
- What is POSIX thread API? How is it used?

switching the address space

Posix thread API (pthread_create)

```
int pthread_create( pthead_t *tid, pthread_attr_t *attr,
                    void * (*thfunc) (void*), void *arg);
```

- Creates a thread with "tid" as its handle and the thread starts executing the function pointed to by the "thfunc" argument
- A single argument (of type void *) can be passed to the thread
- Thread attribute can be used to control the thread behavior e.g., stack size, stack address etc. Passing NULL sets the defaults
- Returns 0 on success.
- Thread termination: return from thfunc, pthread_exit( ) or pthread_cancel( )
- In Linux, pthread_create and fork implemented using clone( ) system call

Posix thread API (pthread_join)

int pthread_join(pthead_t tid, void **retval)

- This call waits for the thread with handle “tid” to finish
- The return value of the thread is captured using the “retval” argument
  - The thread must allocate the return value which is freed after the process joins
- Invoking pthread_join for an already finished thread returns immediately