Threads

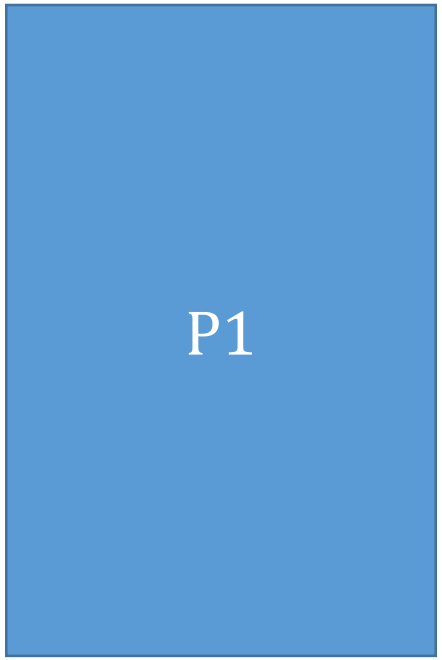
Modern Operating Systems, by Andrew Tanenbaum — Chap 2

Operating Systems: Three Easy Pieces (a.k.a. the OSTEP book) — Chap 26 & 27

"Advanced Programming in Unix Environment" by Richard Stevens
http://www.kohala.com/start/apue.html — Chap 11

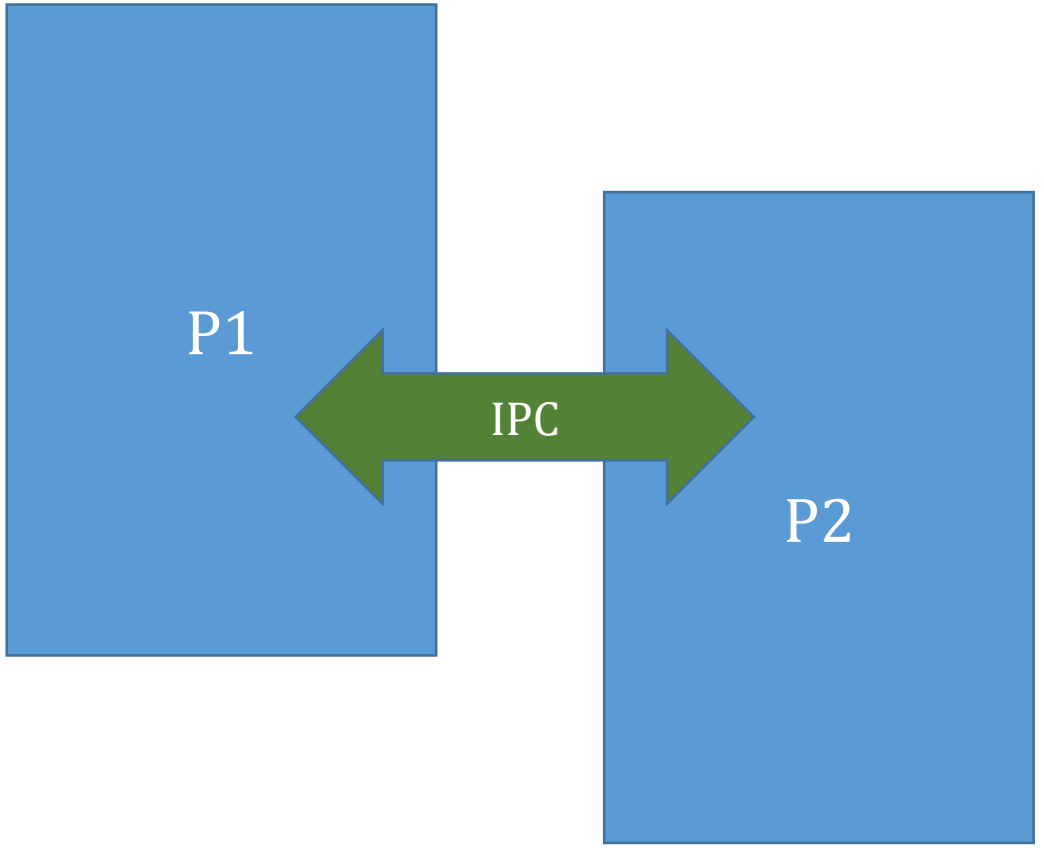
No Concurrency

- If you want to do 1 task, start one process



Concurrency : Option 1- 2 Processes

- If you want to do 2 tasks, start two processes
  - Problem: fork(…) is expensive
  - Problem: "cold start" penalty
  - Problem: If the tasks interact (e.g. synchronization, data passing, etc.) Need IPC
    - IPC is difficult and expensive
    - IPC requires OS resources (kernel system calls), which are expensive
    - Shared memory difficult to set up



"Concurrency" : Option 2- Event Driven

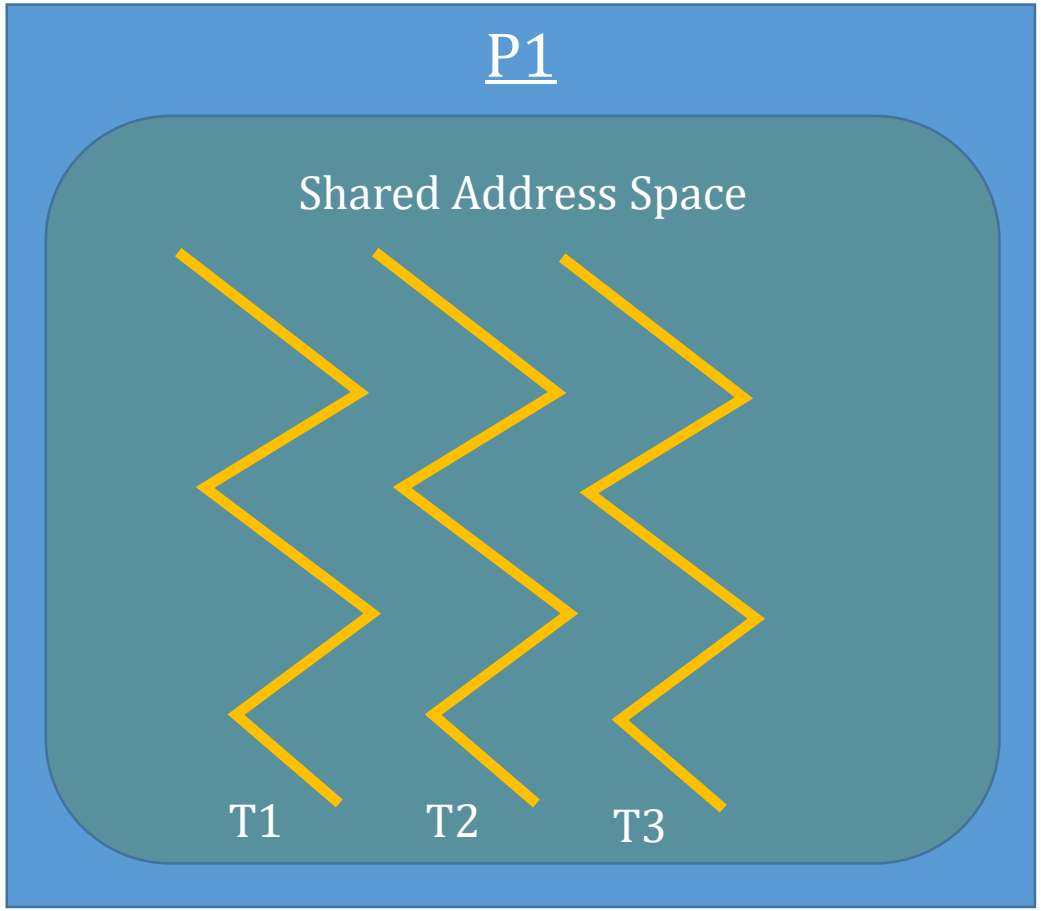
- Make one process do everything
- Busy loop polls for events and performs tasks
- Length of event loop determines response latency
- Long task dominates resources
- Stateful event processing complicates code
  - What if i'th occurrence of task *n* affects the j'th occurrence?

P1

```
while(1) {
    if (event1) do task1;
    if (event2) do task2;
    ...
    if (eventn) do taskn;
}
```

Concurrency : Option 3 - Threads

- Single process
- Single address space
  - code, heap, static data
- Multiple "threads" of execution
- Each thread has it's own:
  - Instruction Pointer
  - Stack / Stack pointer
  - Registers



More on Thread Uniqueness

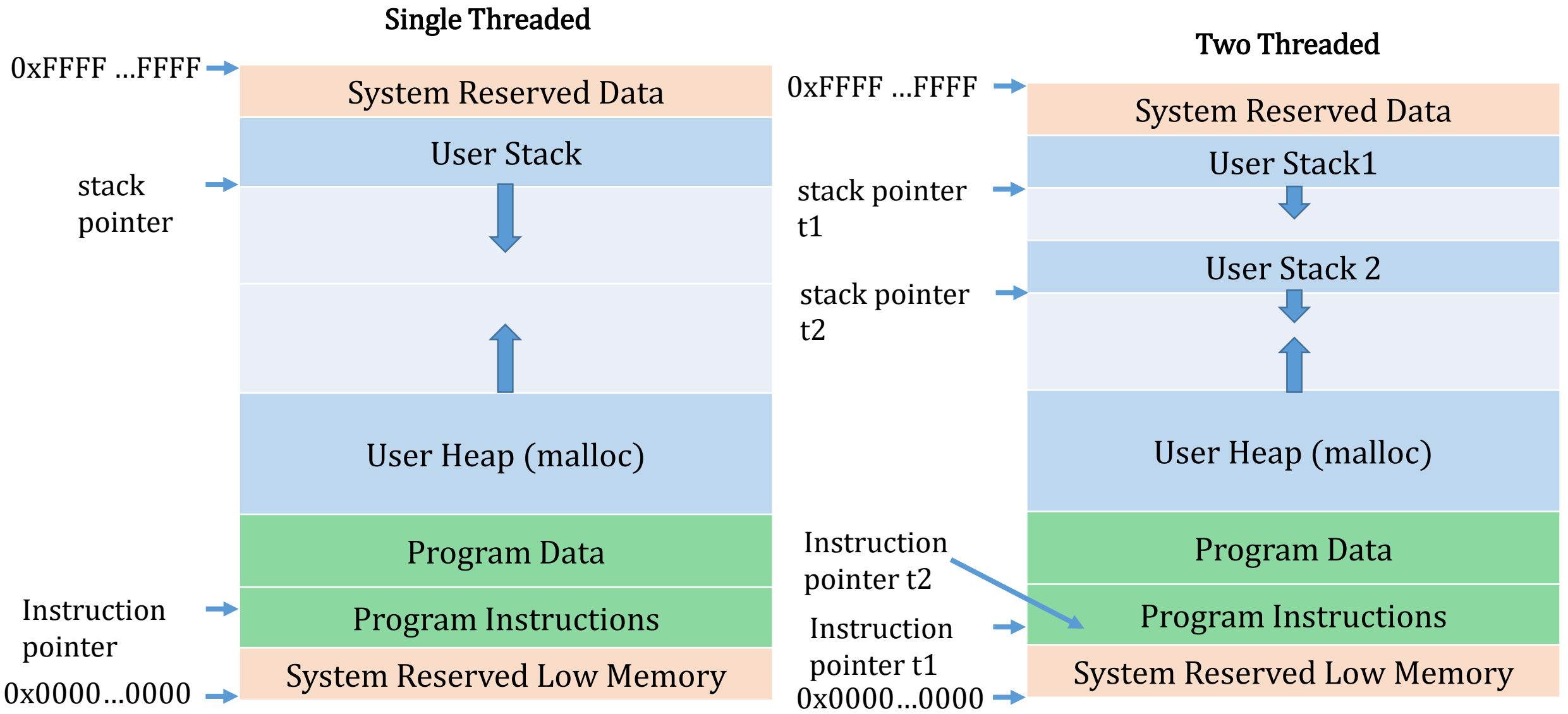
Threads Share

- File description table
  - Open descriptors – files, devices, etc.
- Signals and Signal Handlers

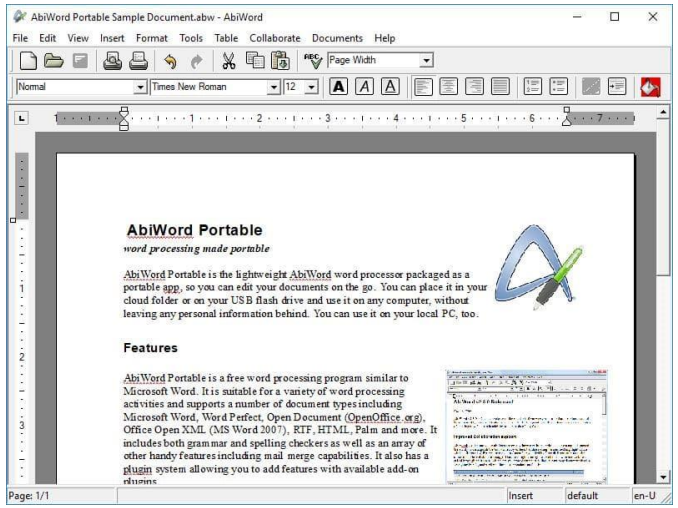
Threads have their own

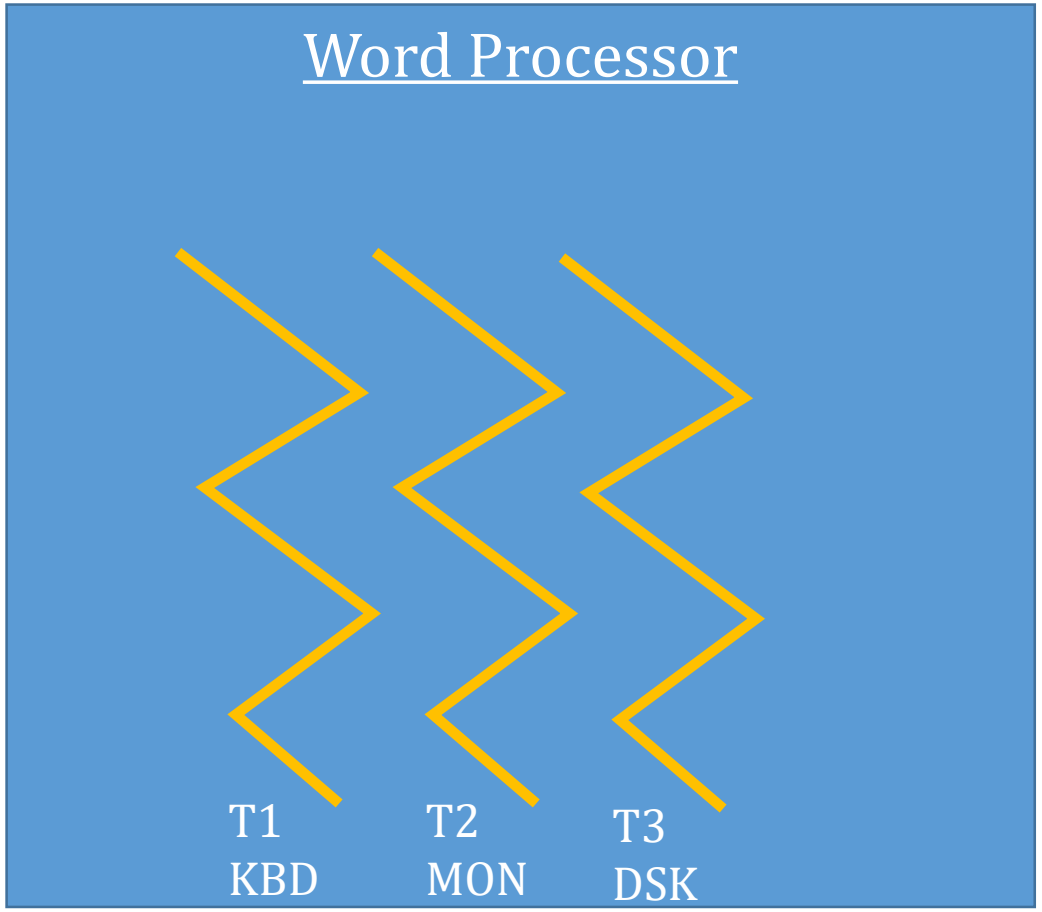
- Thread ID
- errno
- Priority

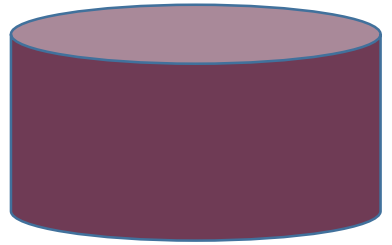
Address space layout



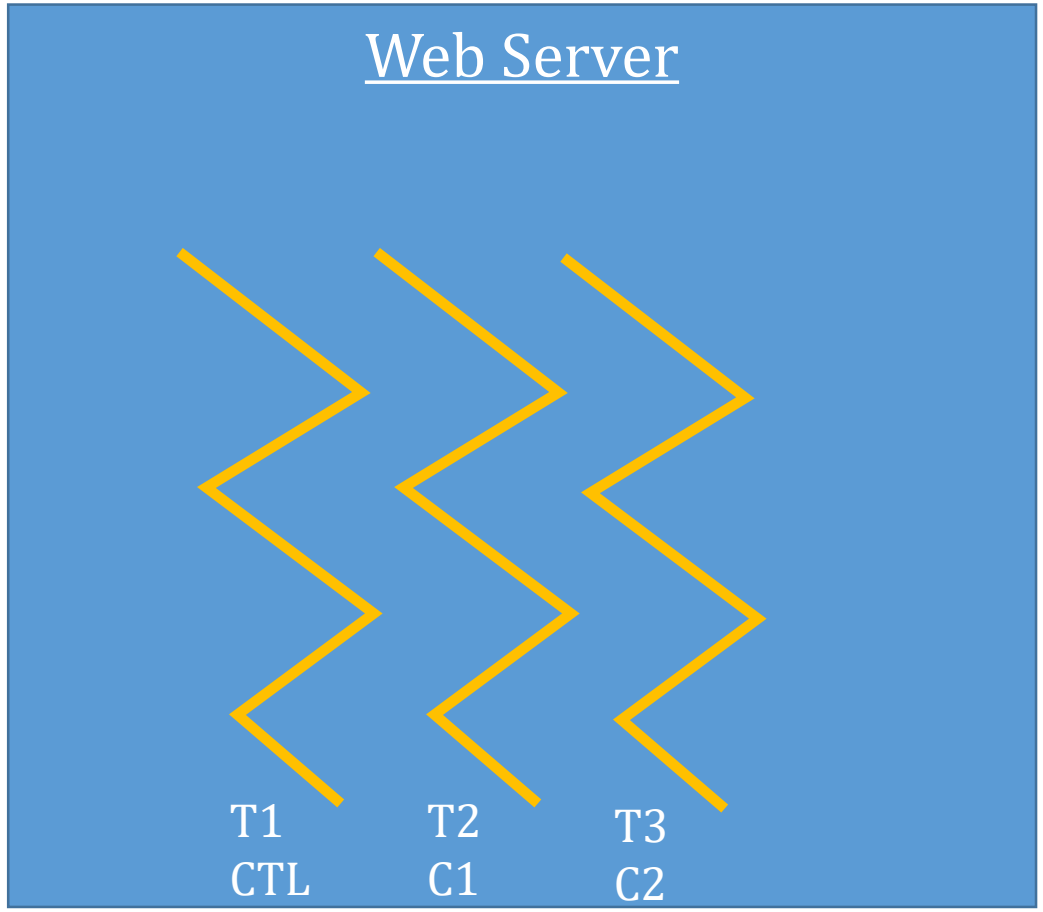
Threaded Process Example







Threaded Process Example



- CTL Thread manages connections
  - Accepts new connections
  - Starts a thread for each connection
- Cx threads manage a single connection

Advantages of Threads

- Low cost context switching between threads
  - No memory swapping
- No inter-process communication required
  - No data transfer required between threads
  - But inter-thread coordination still required!
- Threads are easy to pre-empt
  - Compare to event driven programming with long running task
  - Long running threads are not a problem
- Threads might exploit parallelism (More later)

Disadvantages of Threads

- Shared State
  - Global variables are shared between threads
  - Accidental changes can cause errors
- Threads don't mix with signals
  - Common signal handler for all threads in a process
  - Which thread to signal? Everybody!
  - Very difficult to program correctly!
- Lack Robustness – Crash in one thread brings all threads down
- Thread Safety in user code and in libraries

Thread Safety

- Guarantee that multiple copies of code can run concurrently without error
- Requires variables to be local to an invocation of the code
- Requires that the state of returned values must be immutable
- If shared state is required
  - Access to that memory must be controlled
    - One thread cannot start working on that object until the other is finished
  - Certain operations must be atomic
    - No other thread can access the state until the operation is fished
    - e.g. "Test and Set"

Kernel vs. User level threads

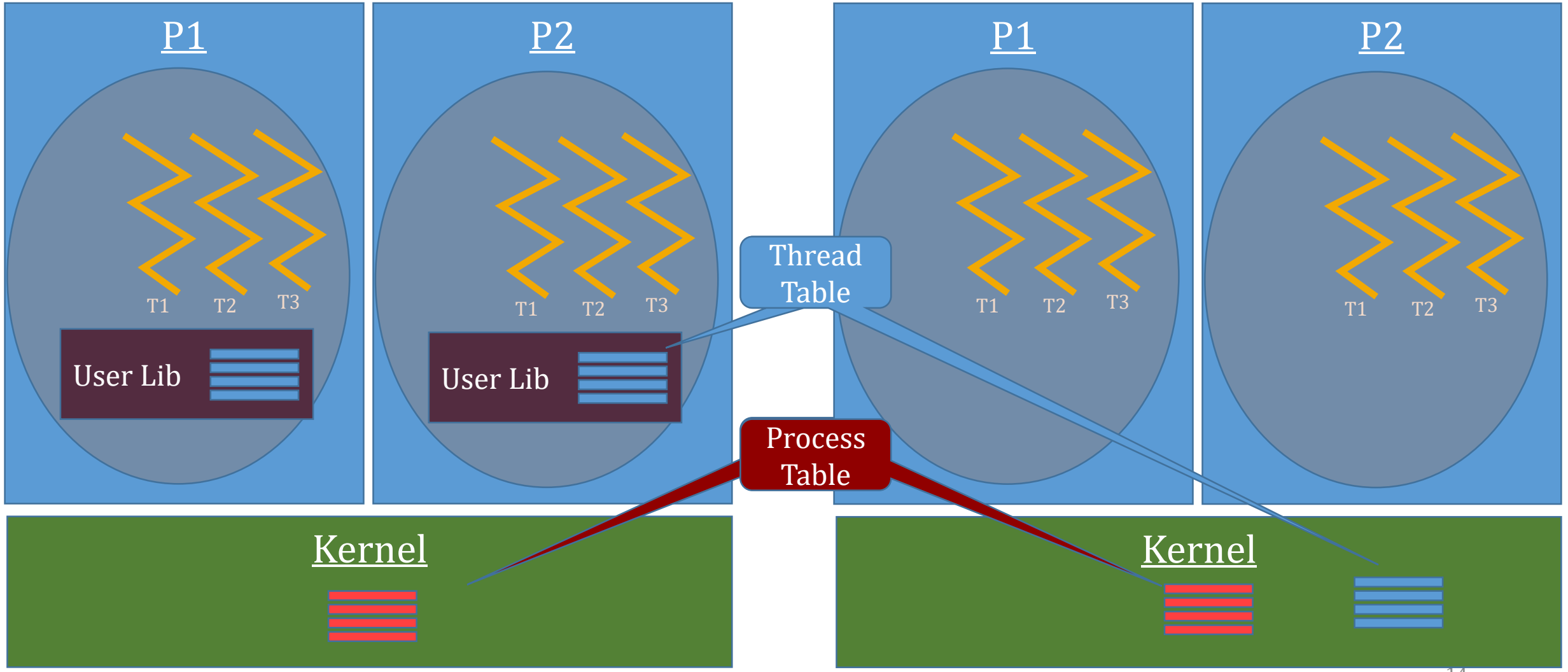
User Level Threads

- Threads managed by system libraries
- OS kernel does not recognize threads
- Threads execute when process is scheduled

Kernel Level Threads

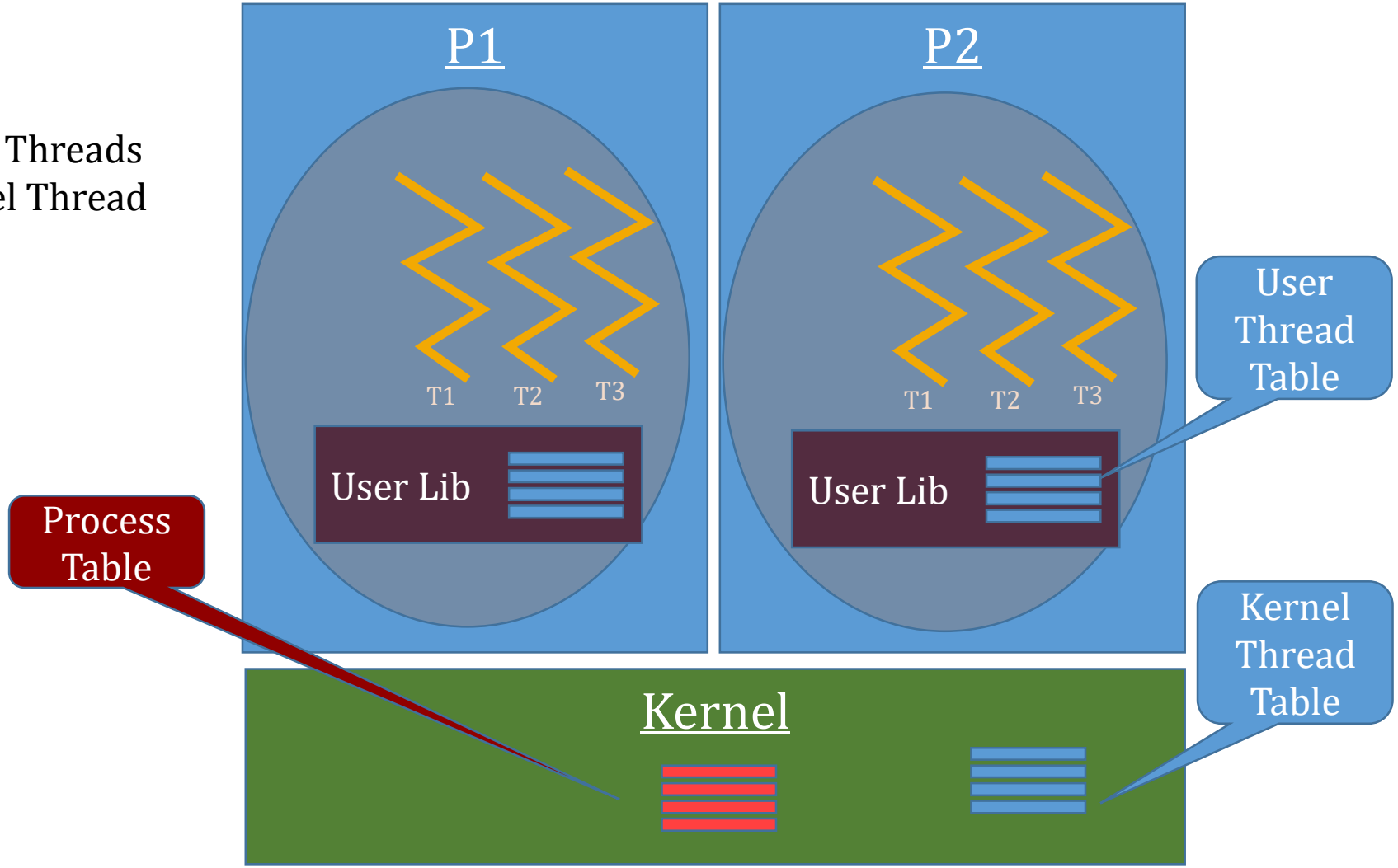
- OS kernel provides multiple threads per process
- Each thread is scheduled independently by the kernel's CPU scheduler

Kernel vs. User Level Threads



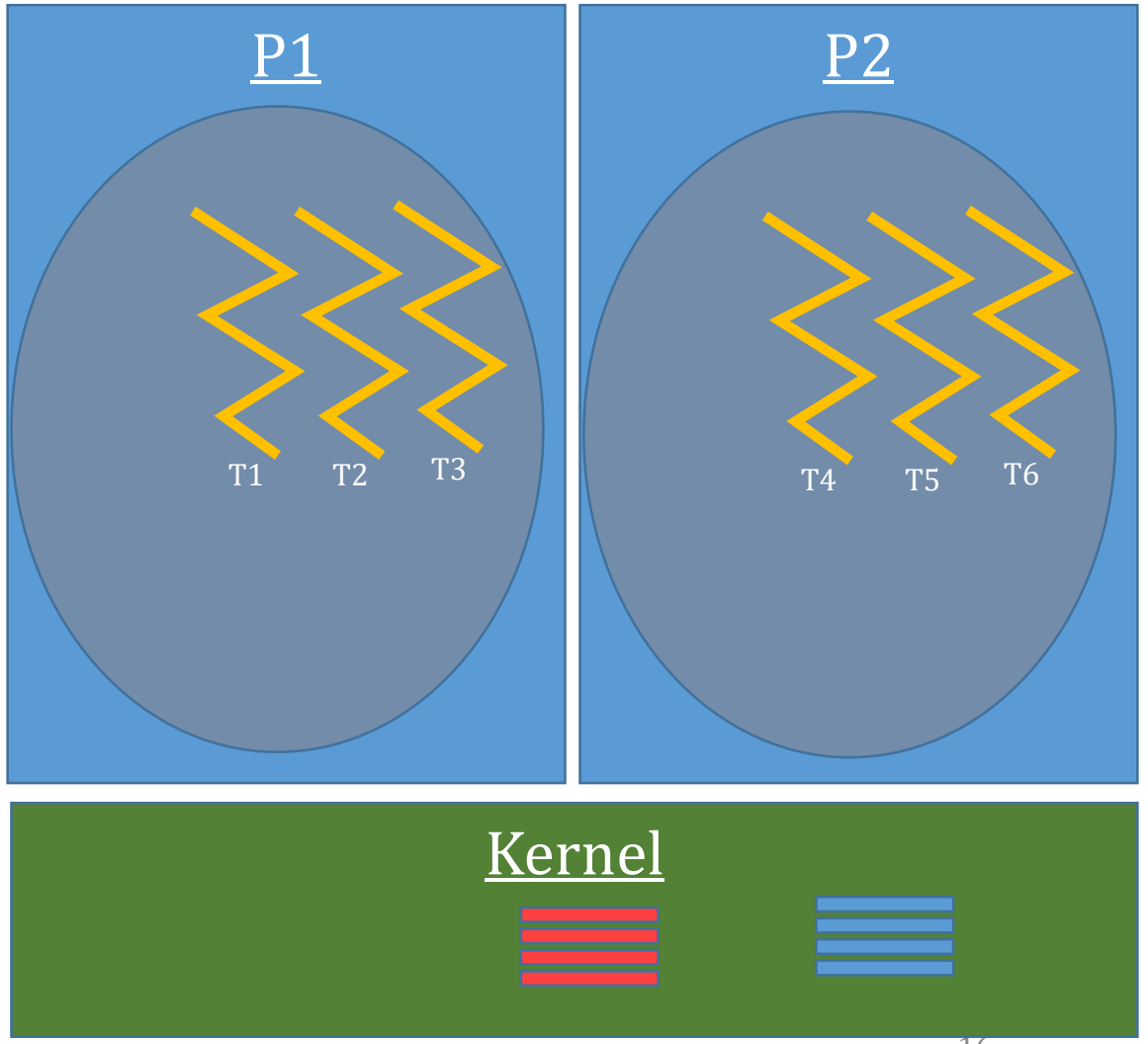
Kernel and User Hybrid

Multiple User Level Threads on each Kernel Level Thread



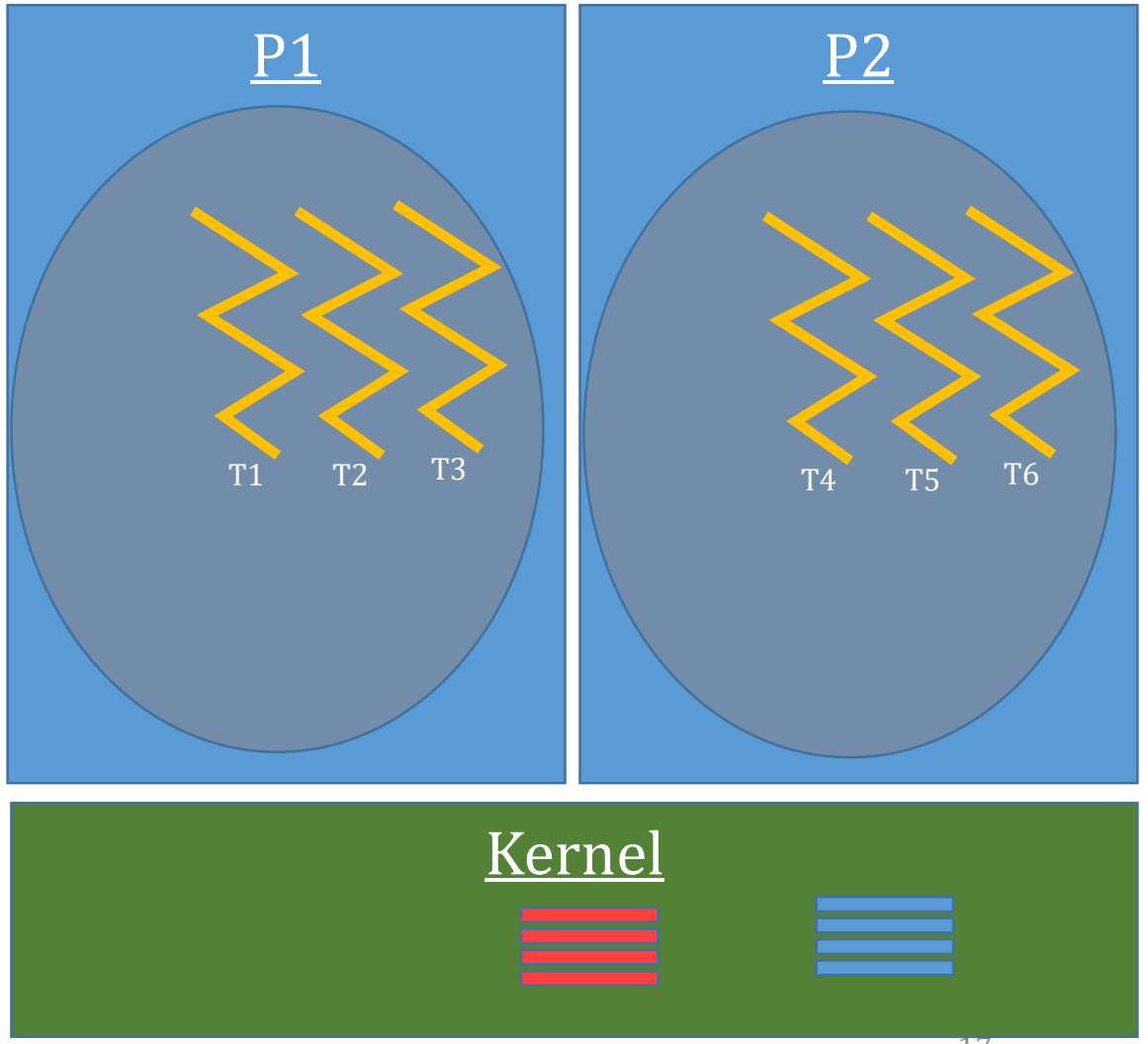
Local Thread Scheduling

- Process gets a time slice
  - Threads in that process get a portion of that timeslice
- May be used by either
  - User level threads OR
  - Kernel level threads
- Scheduling requires only local knowledge of threads within a single process
- Can have: T1, T2, T3, T1, …
- Cannot have: T1, T4, T2, T5, …



Global Thread Scheduling

- Thread gets a time slice
  - Thread may be chosen from any process
  - No notion of process time slice
- Only Kernel Threads
  - Requires knowledge of threads in other processes
- Can have: T1, T2, T3, T1, …
- Can have: T1, T4, T2, T5, …



Thread creation and Termination

Start thread with:

```
int pthread_create(
  pthread_t * thread,
  pthread_attr_t * attr,
  void * (*start_routine)(void *),
  void * arg
);
```

End Thread with:

- return from `start_routine`

Or

```
void pthread_exit(void * status)
```

Wait for thread termination:

```
int pthread_join(pthread_t thread, void **retval);
```

- Similar to waitpid

Example pthread program

```
// shared counter to be incremented by each thread
int counter = 0;

int  main() {
   pthread_t tid[N];
   for (int i=0;i<N;i++) {
      pthread_create(&tid[i], NULL, thread_func, NULL);
   }
   for(int i=0;i<N;i++)  /* wait for child thread */
      pthread_join(tid[i], NULL);

   printf(
"After %d threads incremented counter, value is %d\n",
         N,counter);
   return 0;
}
```

```
void *thread_func(void *arg) {
    /* unprotected code – race condition*/
    if (counter<10) {
       printf(
"Thread: incrementing counter = %d\n",
          counter);
       counter = counter + 1;
    }
    return NULL; // thread dies upon return
}
```

Example pthread output (N=25)

```
./pthread_race
Thread: incrementing counter = 0
Thread: incrementing counter = 0
Thread: incrementing counter = 0
Thread: incrementing counter = 0
Thread: incrementing counter = 0
Thread: incrementing counter = 1
Thread: incrementing counter = 2
Thread: incrementing counter = 5
Thread: incrementing counter = 5
Thread: incrementing counter = 7
Thread: incrementing counter = 8
Thread: incrementing counter = 9
Thread: incrementing counter = 9
After 25 threads…, value is 13
```

pthread synchronization utilities

Mutexes

```
pthread_mutex_init(…)
pthread_mutex_lock(…)
pthread_mutex_unlock (…)
pthread_mutex_trylock (…)
```

Condition Variables

```
pthread_cond_wait (…)
pthread_cond_signal (…)
pthread_cond_broadcast (…)
pthread_cond_timedwait (…)
```

Example protected pthread program

```
int counter = 0;
pthread_mutex_t counter_mutex;

int  main() {
   pthread_t tid[N];
   pthread_mutex_init(&counter_mutex, NULL);
   for (int i=0;i<N;i++) {
      pthread_create(&tid[i], NULL, thread_func, NULL);
   }
   for(int i=0;i<N;i++)  /* wait for child thread */
      pthread_join(tid[i], NULL);
   pthread_mutex_destroy(&counter_mutex);

   printf(
"After %d threads incremented counter, value is %d\n",
      N,counter);
   return 0;
}
```

```
void *thread_func(void *arg) {
   pthread_mutex_lock(&counter_mutex);
    if (counter<10) {
       printf("Thread: incrementing counter = %d\n",
         counter);
       counter = counter + 1;
    }
    pthread_mutex_unlock(&counter_mutex);
    return NULL; // thread dies upon return
}
```

Example protected pthread output (N=25)

```
./pthread_race_protected
Thread: incrementing counter = 0
Thread: incrementing counter = 1
Thread: incrementing counter = 2
Thread: incrementing counter = 3
Thread: incrementing counter = 4
Thread: incrementing counter = 5
Thread: incrementing counter = 6
Thread: incrementing counter = 7
Thread: incrementing counter = 8
Thread: incrementing counter = 9
After 25 threads incremented counter, value is 10
```