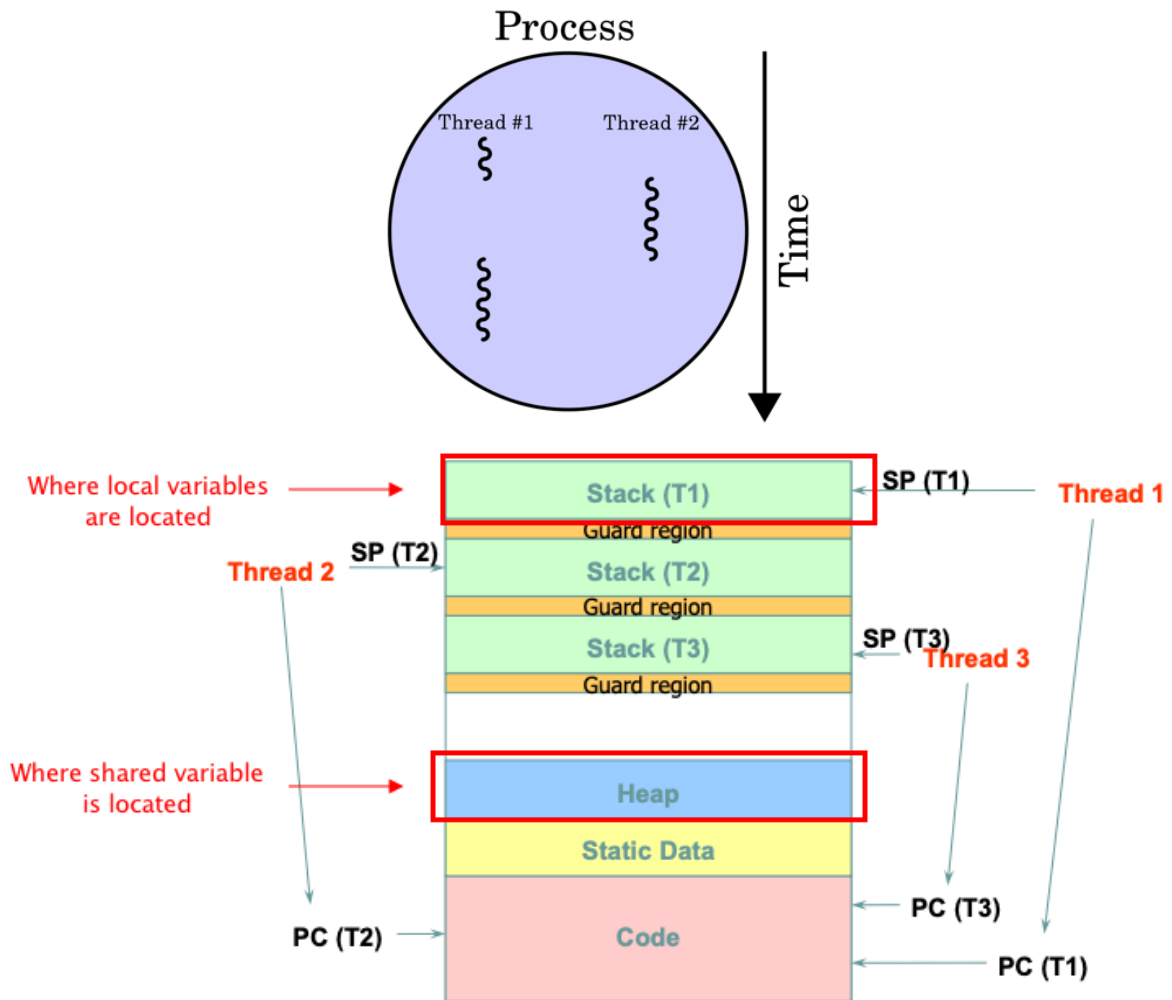


1 Critical Section

- Is a piece of code that accesses a *shared* resource, usually a variable or data structure

2 Thread

- Is a lightweight process that can be managed independently by a scheduler ^[4]
- Improves the application performance using parallelism. (e.g. peach)



- A thread is bound to a single process
- A process can have multiple threads
- Has two types
 - **User-level Threads:**

- * Are implemented by users and kernel is not aware of the existence of these threads
- * Are represented by a program counter(PC), stack, registers and a small process control block
- * Are small and much faster than kernel level threads
- **Kernel-level Threads:**
 - * Are handled by the operating system directly
 - * Thread management is done by the kernel
 - * Are slower than user-level threads

3 Thread API

- pthread_create

- **syntax:**

```

1 int pthread_create(pthread_t *thread,
2                   const pthread_attr_t *attr,
3                   void * (*start_routine) (void*),
4                   void * arg)

```

- * thread
 - is a pointer to a structure of type pthread_t
- * attr
 - is used to specify any attributes this thread might have
 - is initialized with a separate call pthread_attr_init()
 - set default by passing NULL
- * (start_routine)
 - means which function this thread should start running in?
 - setting void pointer (void *) as an argument to function start_routine allows us to pass in any type of argument
 - setting void pointer (void *) as return type allows us to return any type of result
- * args
 - is where to pass the arguments for the function pointer ((start_routine))

Example

```

1  #include <stdio.h>
2  #include <pthread.h>
3
4  typedef struct {
5      int a;
6      int b;
7  } myarg_t;
8
9  void *mythread(void *arg) {
10     myarg_t *args = (myarg_t *) arg;
11     printf("%d %d\n", args->a, args->b);
12     return NULL;
13 }
14
15 int main(int argc, char *argv[]) {
16     pthread_t p;
17     myarg_t args = { 10, 20 };
18
19     int rc = pthread_create(&p, NULL, mythread, &args);
20     ...
21 }

```

Struct is copied here

Argument is initialized here

- `pthread_cond_wait`

- Puts the calling thread to sleep (a blocked state)
- Waits for some other thread to signal it

Example

```

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (ready == 0)
    pthread_cond_wait(&cond, &lock);
pthread_mutex_unlock(&lock);

```


Puts calling thread cond to sleep

- `pthread_cond_signal`

- Is used to unblocks at least one of the threads that are blocked on the specified condition variable cond

Example

```
Pthread_mutex_lock(&lock);  
ready = 1;  
Pthread_cond_signal(&cond);  
Pthread_mutex_unlock(&lock);
```



Wakes a thread that's been put to sleep
on cond variable

4 Condition Variable

- is an explicit queue that threads can put themselves on when some state of execution is not as desired (so it can be put to sleep)
- when states are changed, one or more of the waiting threads can be awoken and be allowed to continue (done by **signaling** the condition)
- queue is **FIFO**
- `wait()` call is used to put thread to sleep
- `signal()` call is used to awake thread from sleep
- **Syntax (initialization):**

```
pthread_cond_t c = PTHREAD_COND_INITIALIZER
```

Example

```
1  int done = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```



Initialized here

- Syntax (Wait):

`Pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`

Example

```
1  int done = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```


Put thread to sleep until done

- Syntax (Signal):

`Pthread_cond_signal(pthread_cond_t *c)`

Example

```
1  int done = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```



Awake thread here

5 Spinlock

- Is the simplest lock to build
- Uses a lock variable
 - 0 - (available/unlock/free)
 - 1 - (acquired/locked/held)
- Has two operations
 1. acquire()

```
boolean test_and_set(boolean *lock)
{
    boolean old = *lock;
    *lock = True;
    return old;
}

boolean lock;

void acquire(boolean *lock) {
    while(test_and_set(lock));
}
```

2. release()

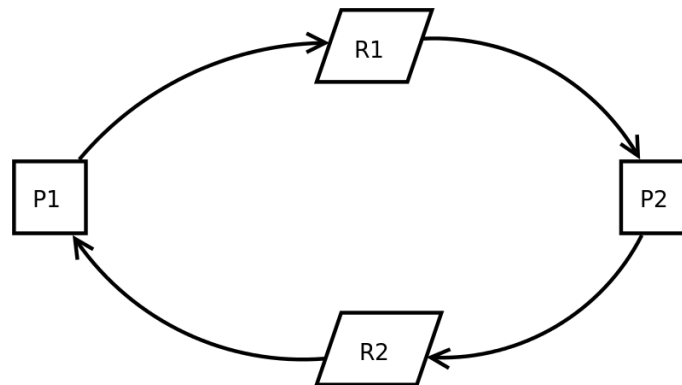
```
void release(boolean *lock) {
    *lock = false;
}
```

- Allows a single thread to enter critical section at a time
- Spins using CPU cycles until the lock becomes available.
- May spin forever

6 Livelock

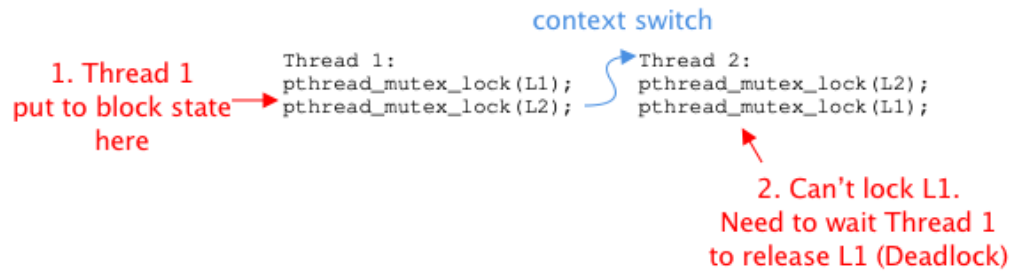
- Two or more threads repeatedly attempting this code over and over (e.g acquiring lock), but progress is not being made (e.g acquiring lock)
- Solution: Add a random delay before trying again (decrease odd of livelock)

7 Deadlock



- Is a state in which each member of a group is waiting for another member including itself, to take action (e.g. releasing lock)
- Conditions for Deadlock (All four must be met)
 - **Mutual Exclusion**
 - * Occurs when threads claim exclusive control of resources that they require (e.g. thread grabbing a lock)
 - **Hold-and-wait**
 - * Occurs when threads hold resources allocated to them (e.g. locks that they have already acquired) while waiting for additional resources (e.g. locks that they wish to acquire)
 - **No Preemption**
 - * Occurs when resource cannot be forcibly removed from threads that are holding them
 - **Circular Wait**
 - * Occurs when there exists a circular chain of threads such that each threads hold one or more resources (e.g. locks) that are being requested by the next thread in the chain.

Example



- **Preventions**

- **Circular Wait**

- * Write code such that circular wait is never induced
- * Is the most practical prevention technique
- * Requires deep understanding of the code base
- * **Total Ordering** (Most straightforward)

Example

Given two locks in the system (L1 and L2), always acquire L1 before L2

- * **Partial Ordering** (Applied to complex systems)

Example

Memory mapping code in Linux (has then different groups).

(Simple) `i_mutex` before `i_mmap_mutex`

(More complex) `i_mmap_mutex` before `private_lock` before `swap_lock`
before `mapping->tree_lock`

- **Hold-and-wait**

- * Can be avoided by acquiring all locks at once
- * Can be problematic
- * Must know which lock must be held and acquire ahead of time
- * Is likely to decrease concurrency (since all need to be acquired over their needs)

Example

```

1 pthread_mutex_lock(prevention); // begin acquisition
2 pthread_mutex_lock(L1);
3 pthread_mutex_lock(L2);
4 ...
5 pthread_mutex_unlock(prevention); // end

```

] Lock all in
order that doesn't cause deadlock
] unlock in
the same order

– No Preemption

- * Can be avoided by adding code that force unlock if not available

Thread 1

```

1 top:
2   pthread_mutex_lock(L1);
3   if (pthread_mutex_trylock(L2) != 0) {
4     pthread_mutex_unlock(L1);
5     goto top;
6   }

```

1. Check if L2 is locked or not available

2. Unlock L1 forcibly (to avoid deadlock)

Thread 2

```

1 top:
2   pthread_mutex_lock(L2);
3   if (pthread_mutex_trylock(L1) != 0) {
4     pthread_mutex_unlock(L2);
5     goto top;
6   }

```

- * pthread_mutex_trylock tries to lock the speicied mutex.
- * pthread_mutex_trylock returns 0 if lock is available
- * pthread_mutex_trylock returns the following error if occupied

EBUSY - Mutex is already locked

EINVAL - Is not initialized mutex

EFAULT - Is in valid pointer

- * May result in **live lock**

– Mutual Exclusion

- * Idea: Avoid the mutual exclusion at all
- * Use **lock-free**/**wait-free** approach: building data structures in a manner that does not require explicit locking using hardware instructions

Example

```

1 int CompareAndSwap(int *address, int expected, int new) {
2   if (*address == expected) {
3     *address = new;
4     return 1; // success
5   }
6   return 0; // failure
7 }

```

- Avoidance

– Banker's Algorithm

8 Process

- Is a program in execution
- Is named by it's process ID or PID
- Can be described by the following states at any point in time
 - Address Space
 - CPU Registers
 - Program Counter
 - Stack Pointer
 - I/O Information

(wait. this is PCB)

- Exists in one of many different **process states**, including
 1. Running
 2. Ready to Run
 3. Blocked
 - Different events (Getting Scheduled, descheduled, or waiting for I/O) transitions one of these states to the other

References

- 1) Coding Horror, Understanding User and Kernel Mode, [link](#)
- 2) Kansas State University, Basics of How Operating Systems Work, [link](#)
- 3) Kansas State University, Glossary, [link](#)
- 4) Tutorials Point, User-level threads and Kernel-level threads, [link](#)
- 5) Science Direct, Scheduling Policy, [link](#)
- 6) Guru 99: What is CPU Scheduling?, [link](#)
- 7) Wikipedia: Round-robin Scheduling, [link](#)
- 8) Kansas State University, The Process Scheduler, [link](#)