OPERATING

SYSTEM

Study Notes

Zhijie Xia
Study Notes
`github.com/zhijie-os`

Contents

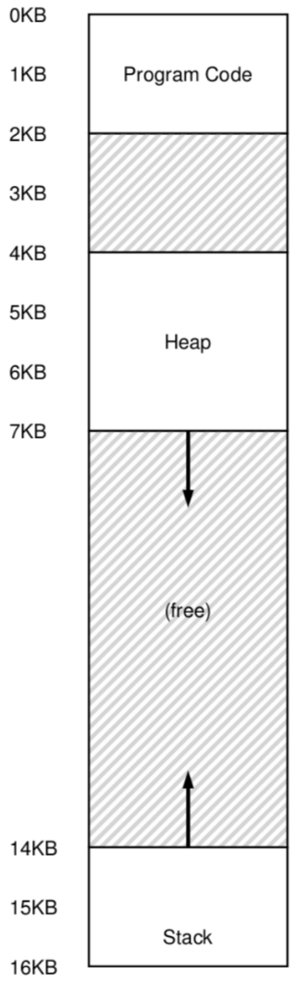
Chapter 1

Virtualization

1.1 Segmentation And Paging

1.1.1 Segmentation

Segmentation is compiler's view about memory.



In canonical address space, we have three logically-different segments:

1. Code/Text
2. Heap
3. Stack

Base/Bound registers

Each segments would have a pair of base and bound registers.
We would use base and bound/limit registers to translate address.

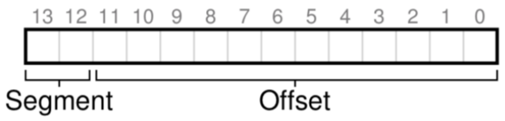
$$\text{Physical Address} = \text{Base Address} + \text{Offset}$$

Bound register is used to check boundary. If offset is greater than bound register value $\Rightarrow$ Segmentation fault.
Note: the stack grows in opposite direction

Referring Segments

We would chop up the address space into two parts:

1. 2 bits: indicate the segments
2. the reset: offset within the segments

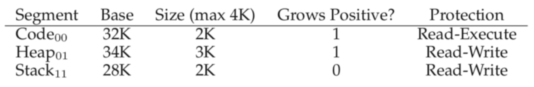


Assume 8-bit address space in use:

1. 00xxxxxx: this is invalid, we would address 1/4 less because we don't use 00.
2. 01xxxxxx: would be in the Code segment.
3. 10xxxxxx: would be in the Heap segment.
4. 11xxxxxx: would be in the Stack segment.

Support for sharing

It is good idea to share the code segment, but how the OS supports it?
There would be protection bits to indicate whether the segment can be used for.



Segment	Base	Size (max 4K)	Grows Positive?	Protection
Code_{00}	32K	2K	1	Read-Execute
Heap_{01}	34K	3K	1	Read-Write
Stack_{11}	28K	2K	0	Read-Write

OS support

So support segmentation, the OS

1. Context Switch: The segment registers must be saved and restored.
2. Support growth or shrinkage of a segment: Management of free space, Compacting physical memory and Free-list management algorithms.

1.1.2 Free Space Management

Low-level Mechanisms

Spliting and coalescing:

Basic Strategies

1.1.3 Advanced Page Tables

1.2 Swapping

1.2.1 Swapping Mechanisms

Use hard disk drive to stash portions of address spaces that currently aren't in great demand, so that we can support programs that take more memories than RAM has.

Swap Space

Swap Space: Reserved space on the disk for moving pages back and forth.
And OS can read from and write to swap space in page-sized units. Also OS needs to know the exact address of the swap space inorder to quickly swap pages.

The Present Bit

Inside of page table entry, there is a present bit.
When the present bit is set, that indicates the page is in the physical memory.
It the present bit is off, the page in not in the memory. When a TLB miss resulting in a page table entry and found the present bit is off, it is a page fault (indicates the demanding page is not in the physical memory).

The Page Fault

The OS would invokes Page-Fault Handler to deal with page fault.

1. The OS would look into PTE(page table entry) to find the address to fetch.
2. After completing disk I/O, the OS updates PTE to mark the page as present.
3. Update the PFN(physical frame number) of the PTE to the in-memory location of the fetched-page.
4. Retry the instruction.

When the memory is full

When page-fault and the memory is full, the OS needs to kick out a page to place a new page in. Therefore, page-replacement policy is needed.

1.2.2 Swapping Policies

Cache Management

Main memory holds some subset of all the pages of ongoing processes $\Rightarrow$ main memory is a cache for virtual memory pages.
The goal is to minimize the number of cache misses.
Average Memory Access Time

$$AMAT = T_M + (P_{miss} \times T_D)$$

Where,

1. T_M: the cost of accessing memory
2. T_D: the cost of accessing disk
3. P_miss: the probability of cache miss

Assume that $T_M = 100$ nanoseconds, $T_D = 10$ milliseconds .
If the hit rate is 0.9, the AMTA would be 100ns + 0.1 · 10ms = 1.0001 ms.
However, when the hit rate is 0.99, the AMTA would be 10.01 microsecs which is 100x faster. That is, the perfomance is heavily based on the hit rate $\rightleftarrows$ swapping policy matters.

The Optimal Replacement Policy

The optimal replacement policy leads to the fewest number of misses overall.

Belady (a person) showed that a simple policy that leads to optimal: The page that would be accessed furthest in the future is the optimal policy (This is very like shortest job first CPU scheduling,i.e, impossible!)

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

First three accesses are misses, and the misses are called **cold-start misses**.

When Access 3 at the first time, the optimal policy decides to evict 2 because 0 and 1 would be accessed before 2.

However, future is unpredicable, an another approach is needed.

FIFO

Old friend, First In First Out.

FIFO is very simple to implement.

Access	Hit/Miss?	Evict	Resulting Cache State	
0	Miss		First-in→	0
1	Miss		First-in→	0, 1
2	Miss		First-in→	0, 1, 2
0	Hit		First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2
3	Miss	0	First-in→	1, 2, 3
0	Miss	1	First-in→	2, 3, 0
3	Hit		First-in→	2, 3, 0
1	Miss	2	First-in→	3, 0, 1
2	Miss	3	First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2

FIFO could often do poorly because it cannot determine the importance of a page.

Random

Simple to implement, and do well if the distribution of accessing page is uniform distributed. However, it is unlikely for accessing page to follow a particular distribtion.

Random is better than FIFO, and a bit worst than optimal.

LRU

If page is accessed in the near past, it is likely to be accesed in near future.

Some historically-based algorithms are used. LFU: Least-Frequently-Used, and LRU: Least-Recently-Used.

Access	Hit/Miss?	Evict	Resulting Cache State	
0	Miss		LRU→	0
1	Miss		LRU→	0, 1
2	Miss		LRU→	0, 1, 2
0	Hit		LRU→	1, 2, 0
1	Hit		LRU→	2, 0, 1
3	Miss	2	LRU→	0, 1, 3
0	Hit		LRU→	1, 3, 0
3	Hit		LRU→	1, 0, 3
1	Hit		LRU→	0, 3, 1
2	Miss	0	LRU→	3, 1, 2
1	Hit		LRU→	3, 2, 1

The LRU policy works well to matching the optimal.

Implement Historical Algorithms

Using LRU, the system needs to count the least- and most-recently used which is a lot of work. Bad implementation would led heavy performance penalty.

Even adding timestamp for every process accessing, it is unlikely to scan the all pages to find the abosulte least recently used page.

Approximating LRU would be a solution:

1. Need a use bit: use bit is set to 1 when the page is accessed.
2. Use a clock algorithm: a clock pointer points to each particular page, if the use bit is 1, the OS clear the use bit and the pointer points to the next page. If found a page with use bit 0, replace it. The worst case is looping through the entire set of pages for one circle.

Dirty Pages

If a page is modified while in the memory, it is dirty. It would cost a lot to evict dirty page since the page must be written to the disk first. Therefore, most algorithms would favor to evict clean pages over dirty pages.

To support the behavior, the hardware includes a modified bit. The bit is set when the page is modified and cleared when written to disk.

Other Policies

Page selection policy determines when to bring a page into the memory. For most pages, OS would use ***demanding paging***, which means the OS brings the page into memory when it is accessed. Or OS would predict which page would be accessed in the future and bring it to the memory, this is called ***prefetching***.

Another policy determines how the OS writes pages out to disk. ***Clustering*** is a behavior that OS buffers the changes and write out to disk in one write.

Thrashing

Thrashing: When the demanding of pages exceed the available physical memory, the system would constantly being paging.

Linux would ran ***out-of-memory killer*** to choose some memory-intensive process and kill them.

Chapter 2

Cucurrency

2.1 Cucurrency and Thread

Thread: very much like process, except threads share the same address space and thus can access the same data.

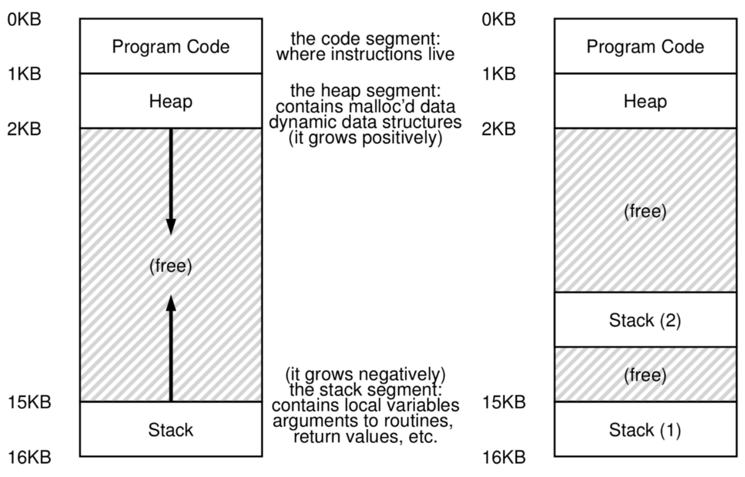
2.1.1 Threads vs Processes

Similarities between Processes and Threads

A thread has a program counter(PC), own set of registers. When switching from thread T1 to thread T2, a context switch must take place. There would be thread control blocks (TCB) like PCB to store the state of each thread.

Difference between Processes and Threads

1. The address space of threads within a single process is the same.
2. Multi-threaded Address Spaces has different structure: one stack per thread called ***thread-local*** storage.


0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows positively)
(it grows negatively) the stack segment: contains local variables arguments to routines, return values, etc.
0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack (2)
(free)
Stack (1)

2.1.2 Benefit of using threads

Parallelism

Run works parallelly.

Avoid Blocking

Avoid blocking program progress due to slow I/O; while one thread in the program waits, the CPU scheduler can switch to other ready threads.

Threading enables overlap of I/O with other activities within a single program.

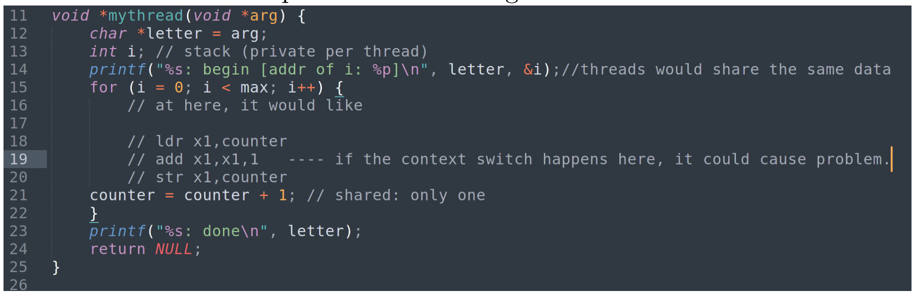
Why not use processes instead?

Threads make it easy to share data, and often used to corporate with other threads to finish tasks.

Processes are more sound choice for logically seperate tasks when little sharing of data structures in memory is needed.

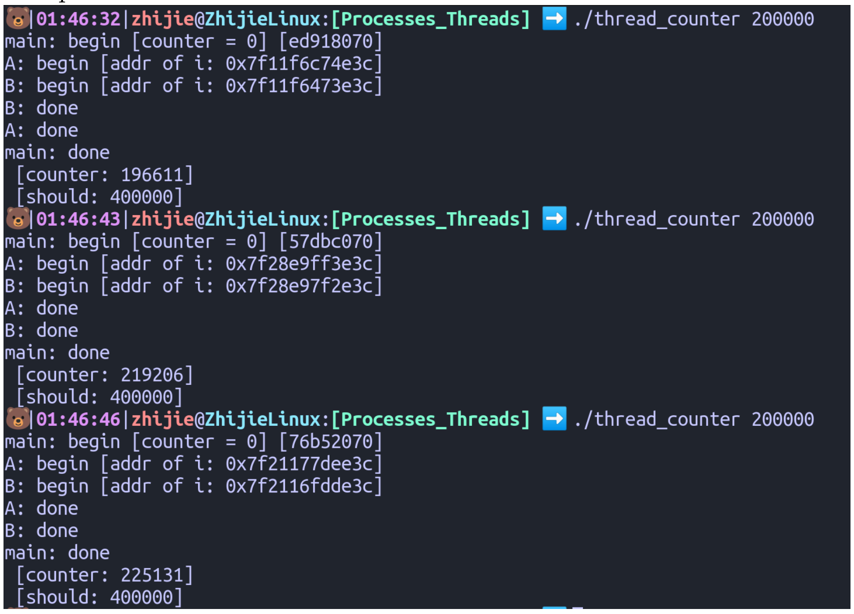
2.1.3 Problem with threads: Race Condition

The execution sequence of threads is indeterministic.
Create two threads to update on the same global variable with the same function.



```c
void *mythread(void *arg) {
    char *letter = arg;
    int i; // stack (private per thread)
    printf("%s: begin [addr of i: %p]\n", letter, &i);//threads would share the same data
    for (i = 0; i < max; i++) {
        // at here, it would like

        // ldr x1,counter
        // add x1,x1,1   ---- if the context switch happens here, it could cause problem.
        // str x1,counter
    counter = counter + 1; // shared: only one
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

The problem can be:



```
🐻|01:46:32|zhijie@ZhijieLinux:[Processes_Threads] ➡ ./thread_counter 200000
main: begin [counter = 0] [ed918070]
A: begin [addr of i: 0x7f11f6c74e3c]
B: begin [addr of i: 0x7f11f6473e3c]
B: done
A: done
main: done
 [counter: 196611]
 [should: 400000]
🐻|01:46:43|zhijie@ZhijieLinux:[Processes_Threads] ➡ ./thread_counter 200000
main: begin [counter = 0] [57dbc070]
A: begin [addr of i: 0x7f28e9ff3e3c]
B: begin [addr of i: 0x7f28e97f2e3c]
A: done
B: done
main: done
 [counter: 219206]
 [should: 400000]
🐻|01:46:46|zhijie@ZhijieLinux:[Processes_Threads] ➡ ./thread_counter 200000
main: begin [counter = 0] [76b52070]
A: begin [addr of i: 0x7f21177dee3c]
B: begin [addr of i: 0x7f2116fdde3c]
A: done
B: done
main: done
 [counter: 225131]
 [should: 400000]
```

Assembly Code

In ARMx8 Assembly:
counter = counter + 1 is equivalent to

1. ldr x1,[counter]

2. add x1, x1, 1

3. str x1, [counter]

The work flow of causing problem

1. Thread A loads counter into x1, say x1=50.
2. Context switch happenes, and switch to Thread B.
3. Now thread B loads counter into x1, x1=50.
4. Thread B increase x1 by 1, x1=51.
5. Thread B stores x1 back to counter, counter=51.
6. Context switch happenes, and switch back to Thread A.
7. Context Switch restores x1 for A,i.e, x1=50. And A won't load counter to x1 again
8. Thread A increase x1 by 1, x1=51.
9. Thread A stores x1 back to counter, counter=51.
10. Thus, counter is set to 51 twice, although it should be 52 after the flow.

Critical Section

Critical Section: A piece of code that accesses a shared variable, and must not be concurrenctly executed by more than one thread.

Mutual Exclusion

Mutual Exclusion: if one thread is executing within the critical section, the others will be prevented from doing so.

Race Condition

Multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading unexpected outcome.

Atomicity

Atomic operation: grouped actions to be executed in one scheduling, i.e, the operation won't be interrupted.

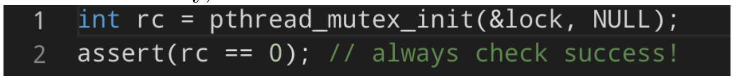
For example, x1 = x1 + x2 could be done in one single step with hardware support, instead of load, add, and store.

It is desired to support atomicity for critical sections.

2.1.4 Thread API

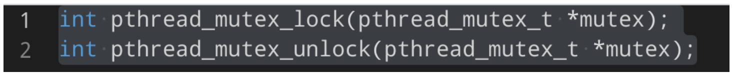
Lock

In POSIX library, a lock needs to be initialized



```c
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

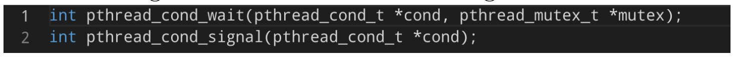
Also, a thread can acquire a lock, and release a lock.



```c
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Condition Variable

Condition variables are useful when some kind of signaling must take place between threads if one thread is waiting for another to do something before it continues.



```c
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

A thread must hold a lock to call either wait() or signal(). $pthread_{c}ond_{w}ait()$, puts the calling thread to sleep. $pthread_{c}ond_{s}ignal()$ awakes the waiting thread.

The reason that $pthread_{c}ond_{w}ait()$ takes two parameter is because it needs to specify which thread to give the lock to. When $pthread_{c}ond_{w}ait()$ the calling thread release the lock and pass it to another thread.

2.1.5 Locks and Building one

Basic

Lock is used around the critical section. It is a global variable that either available or acquired and exactly one thread can hold it at a time.

mutex in POSIX means ***mutual exclusion*** between threads.

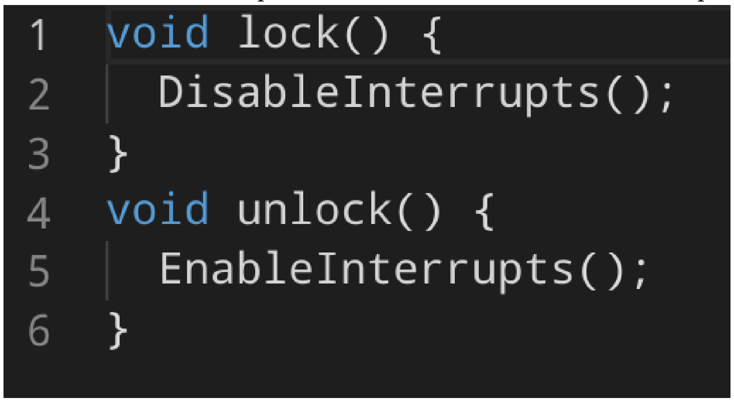
Evaluating Locks

Basic criteria:

1. Mutual exclusion: A lock must provide mutual exclusion, i.e, the lock should preventing multiple threads from entering a critical section.
2. Fairness: Prevent starving a lock.
3. Performance: How many overhead would be added to use the lock.

Lock by Controlling Interrupts

One of the earliest implementation of lock is disable interrupts.

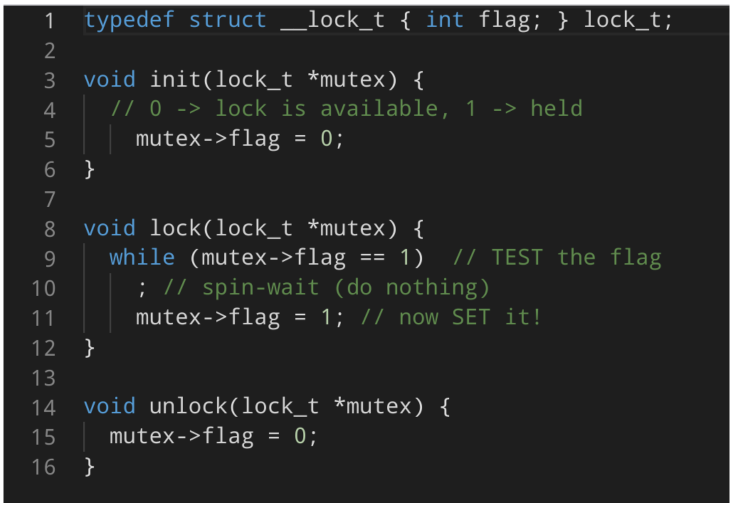


```c
void lock() {
  DisableInterrupts();
}
void unlock() {
  EnableInterrupts();
}
```

This approach works since it assumes mutual exclusion, and it is very simple.
However, it has flaws:

1. Priviledged action: malicious process would disable the interrupts and never enable interrupts again.
2. Interrupts would get lost: for example, I/O interrupts would get lost and some processes which are waiting on those interrupts cannot move forward.
3. Inefficient approach: It is very costly to enable/disable interrupts.
4. No support for multiprocessors.

A Fail Attempt: Using a Flag



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

void init(lock_t *mutex) {
  // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
  while (mutex->flag == 1)  // TEST the flag
    ; // spin-wait (do nothing)
    mutex->flag = 1; // now SET it!
}

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

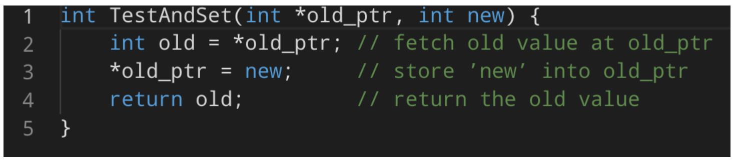
Correctness Problem: Interleaving would give more locks than just one.

Thread 1	**Thread 2**
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Performance Problem: While loops is valid instruction that would use CPU, it is very likely a thread which acquiring the lock spents its timeslot to loop. This behavior is called ***busy-waiting*** or ***spin-waiting***.

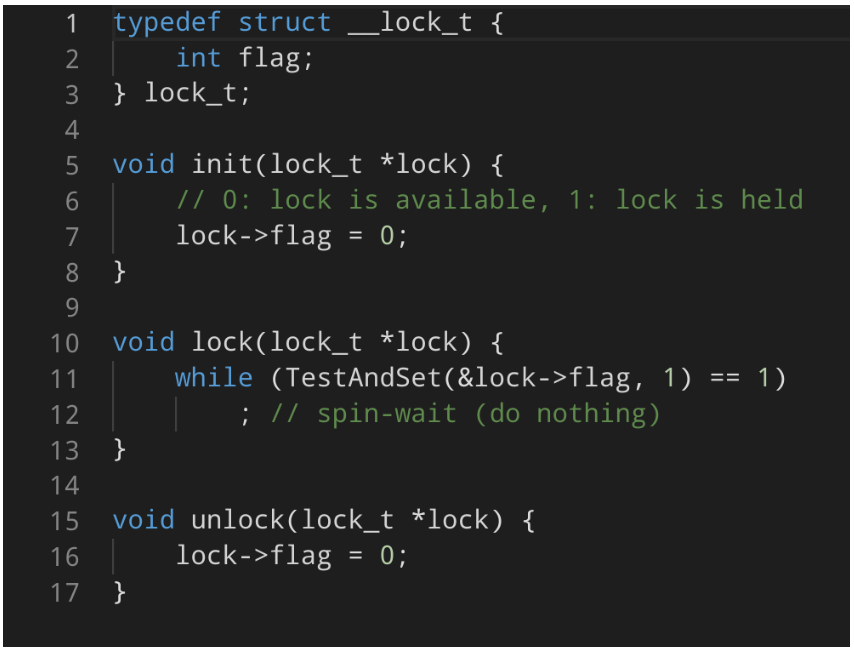
Test-and-Set

Test-and-Set is atomic instruction supported by the hardware. It both gets and sets the value in a register/address.



```c
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

With ***Test-and-Set***, we can build a correct lock.



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

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

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

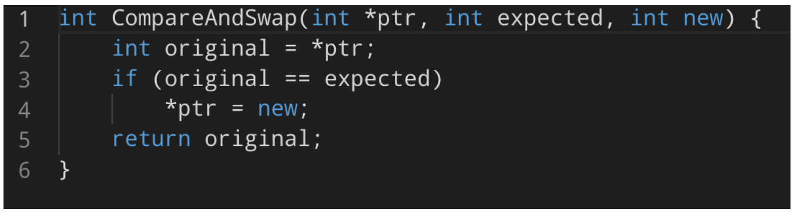
However, the Test-and-Set approach doesn't guarantee

1. Fairness: There is no intelligence invoked to provide fairness.

2. Performance: It is painful on a single CPU. Acceptable on multiple CPUs, because the CPU scheduler would switch the waiting thread out after its timeslot.

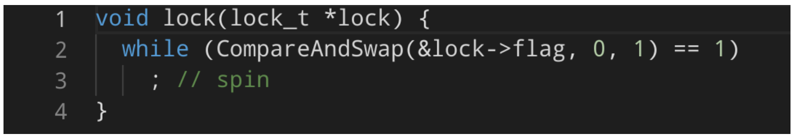
Compare-And-Swap

Compare-And-Swap: Test whether the value equals; is so, update the memory value. Finally, it would return the original value.



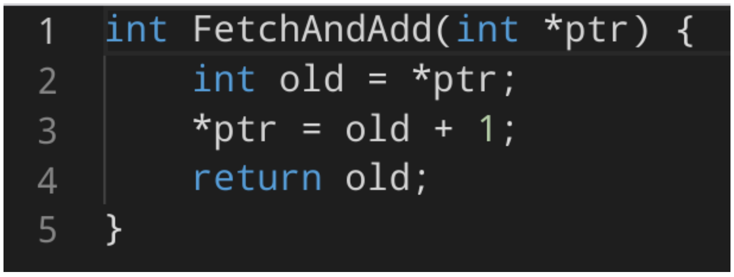
```c
int CompareAndSwap(int *ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}
```

With ***Compare-And-Swap***, it is possible to build a spin clock



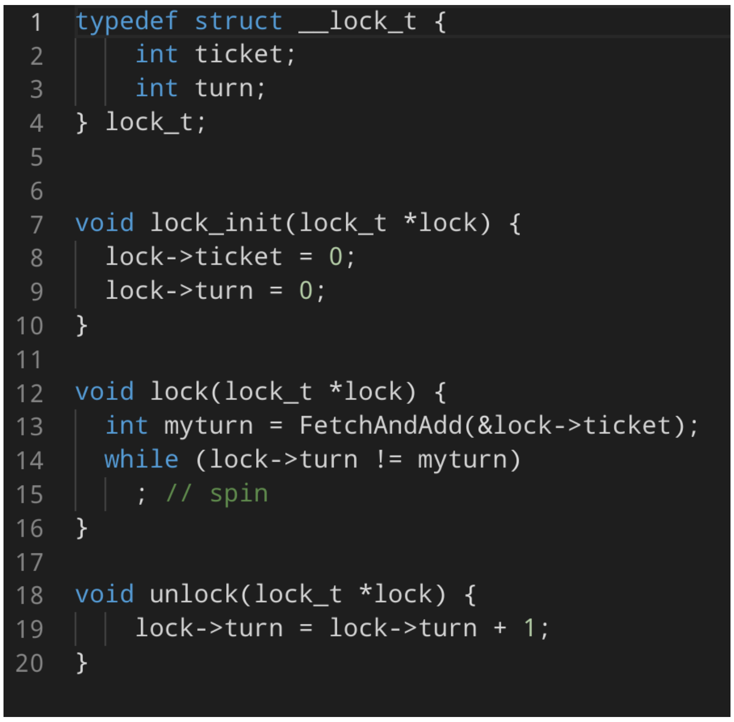
```c
void lock(lock_t *lock) {
  while (CompareAndSwap(&lock->flag, 0, 1) == 1)
    ; // spin
}
```

Fetch-And-Add



```c
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

And a spin clock can be build, a ***ticket lock***



```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;


void lock_init(lock_t *lock) {
  lock->ticket = 0;
  lock->turn = 0;
}

void lock(lock_t *lock) {
  int myturn = FetchAndAdd(&lock->ticket);
  while (lock->turn != myturn)
    ; // spin
}

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

The advantage of ***ticket lock*** is that the approach "remeber" the requests of lock, i.e, it is like to pick a ticket that has number on it. And once the thread picks its ticket, all it needs to do is to wait and be called.

$\Rightarrow$ this approach guarantees fairness.

Other approaches are like fighting with each other for a single ticket.

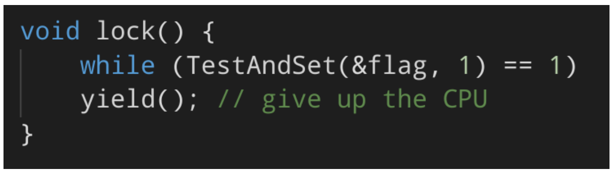
Spin locks and hardware limitation

With the extra hardware supports, we can now build spin locks. However the inefficiency is like a diaster.

Suppose each threads executes the same amount of time, with N threads and a single lock. The actual work done is $\frac{1}{N}$, and $\frac{N-1}{N}$ is useless busy-waiting.

The hardware cannot solve everything, a smart software needs to be introduced in OS.

Just yield, Baby



```
void lock() {
    while (TestAndSet(&flag, 1) == 1)
    yield(); // give up the CPU
}
```

yield() is an operating system primitive which a thread can call when it wants to give up the CPU and let another thread to run, i.e, descheduling the calling thread and move it from running to ready.

Efficiency Problem:Suppose there are N threads and a single lock and the scheduler is taking round robin , N-1 yield would be called and only 1 critical section instruction would be executed. That is still bad.

Fairness Problem: If the scheduler is not using round robin, a thread would be picked consecutive to call yield() which introduce the possibilty of starving a process.

Using Queues: Sleeping Instead of Spinning

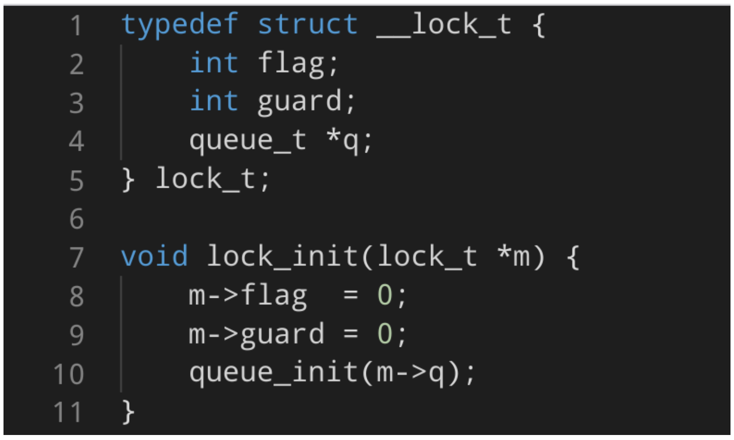
There are some controls needed over which thread next gets to acquire the lock after the current holder release it.

Some OS support is need $\Rightarrow$ A queue to keep track of which threads are waiting to acquire the lock.

park() and ***unpark()***.

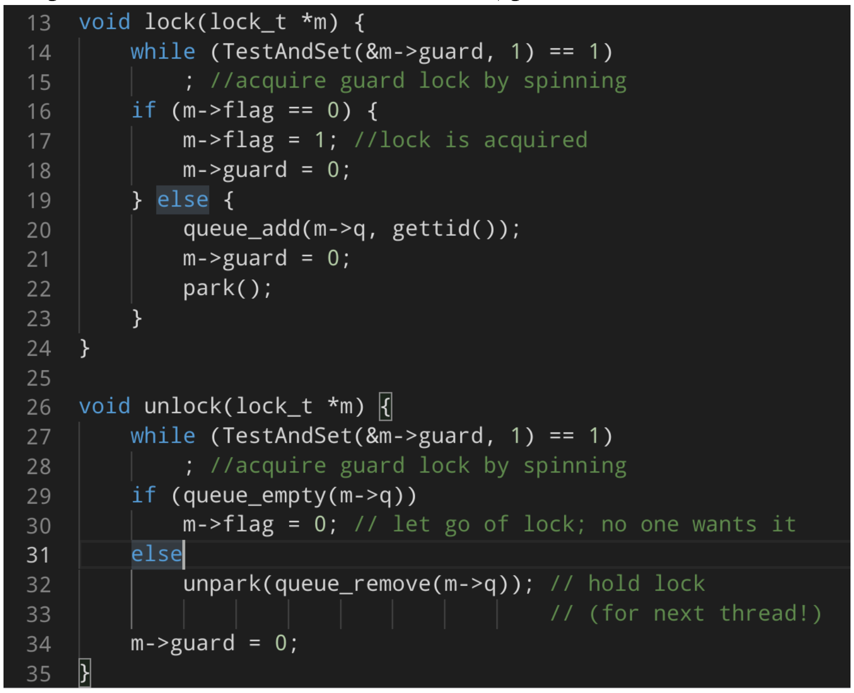
In Solaris: ***park()*** to put a calling thread to sleep and ***unpark(threadID)*** to wake a particular thread as designated by ***threadID***.

When a thread tries to acquire the lock, the OS would park() to put the thread into sleeping and awakes it by calling unpark(threadID) when the lock is free.



```c
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}
```

Flag indicates whether the lock is available, ***guard*** is used to ensure atomicity within the lock()/unlock().



```c
void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; //lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock
                                    // (for next thread!)
    m->guard = 0;
}
```

The guard is like another lock for the lock()/unlock(), one thread needs to hold the guard to acquire the lock or put itself into sleep and release the guard.

Advantage:

1. Small spinning/waste.
2. Fairness by using the queue.

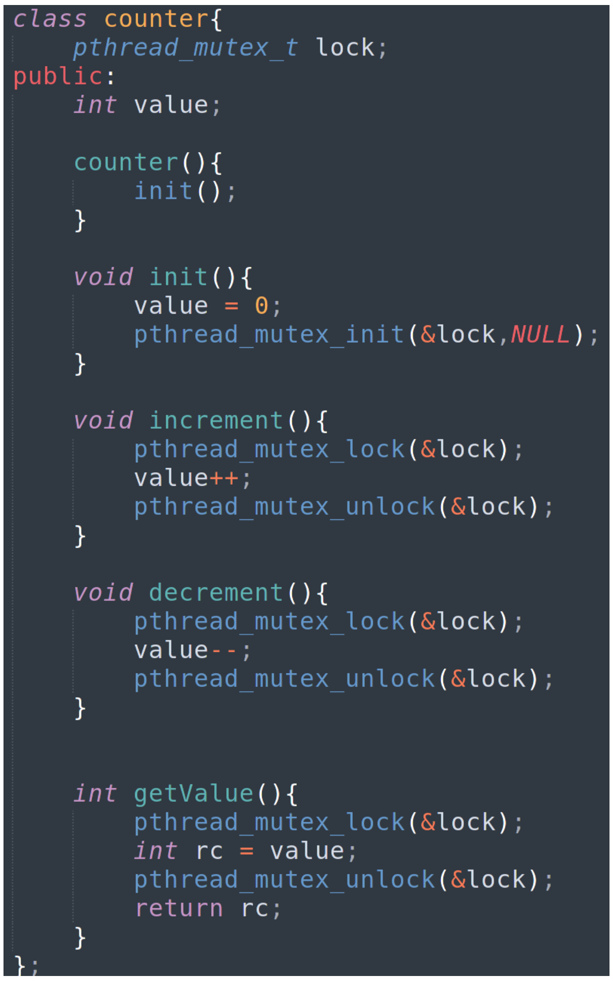
Futex

A linux approach.... Kinda complicated, would take a look when studying linux.

2.1.6 Locked Data Structures

We can use locks in some common data structures to make the structures thread safe.

Cucurrent Counters



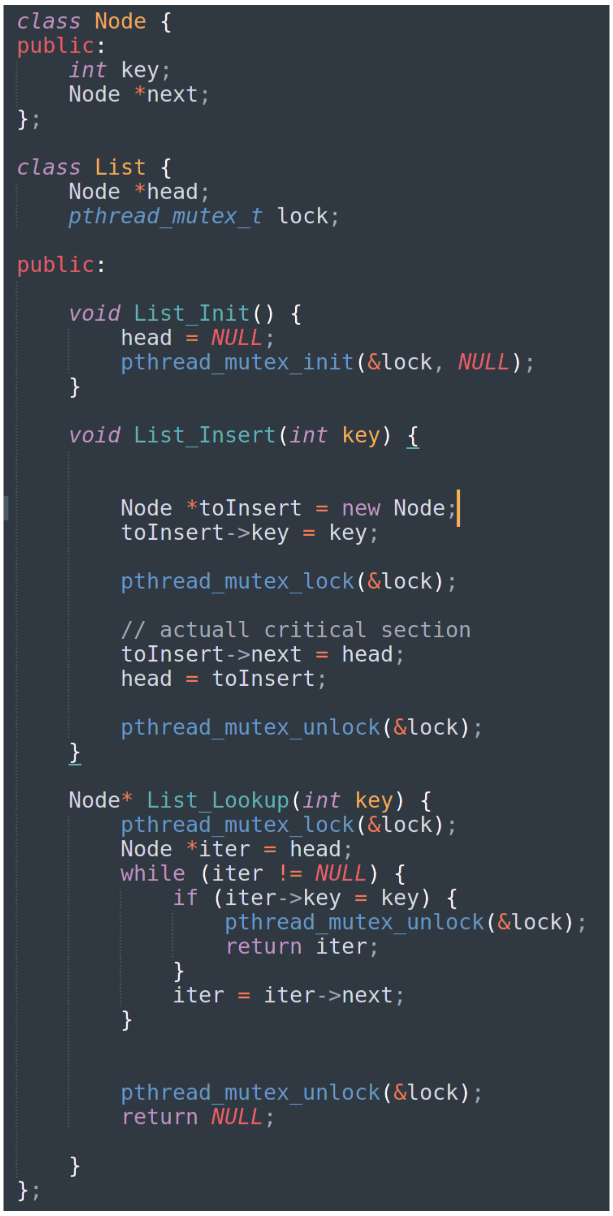
```
class counter{
    pthread_mutex_t lock;
public:
    int value;

    counter(){
        init();
    }

    void init(){
        value = 0;
        pthread_mutex_init(&lock,NULL);
    }

    void increment(){
        pthread_mutex_lock(&lock);
        value++;
        pthread_mutex_unlock(&lock);
    }

    void decrement(){
        pthread_mutex_lock(&lock);
        value--;
        pthread_mutex_unlock(&lock);
    }


    int getValue(){
        pthread_mutex_lock(&lock);
        int rc = value;
        pthread_mutex_unlock(&lock);
        return rc;
    }
};
```

The lock is acquired and released automatically as you call and return from object methods, like a monitor.

However, this approach scales poorly.

Approximate counters works scablely. The idea is to give each thread a local counter, each thread updates on its local counter, and when the counter reaches the threshold. It acquire the global lock and accumulate its local counter to the global counter.

Concurrent Linked Lists



```
class Node {
public:
    int key;
    Node *next;
};

class List {
    Node *head;
    pthread_mutex_t lock;

public:

    void List_Init() {
        head = NULL;
        pthread_mutex_init(&lock, NULL);
    }

    void List_Insert(int key) {


        Node *toInsert = new Node;
        toInsert->key = key;

        pthread_mutex_lock(&lock);

        // actuall critical section
        toInsert->next = head;
        head = toInsert;

        pthread_mutex_unlock(&lock);
    }

    Node* List_Lookup(int key) {
        pthread_mutex_lock(&lock);
        Node *iter = head;
        while (iter != NULL) {
            if (iter->key = key) {
                pthread_mutex_unlock(&lock);
                return iter;
            }
            iter = iter->next;
        }


        pthread_mutex_unlock(&lock);
        return NULL;

    }
};
```

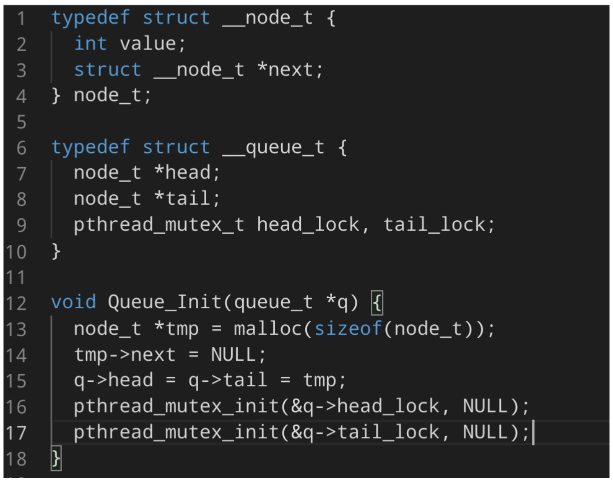
However, this concurrent linked list is not scable.

Hand-over-hand locking : Instead of having a single lock for the entire list, add a lock per node of the list. When traversing the list, the code first grabs the next node's lock and then release the current node's lock.(Potientially Deadlock). Also, practically it is hard to make such a structure faster than the simple single lock approach, as the overheads of acquiring and releasing locks for each node is prohibitive.

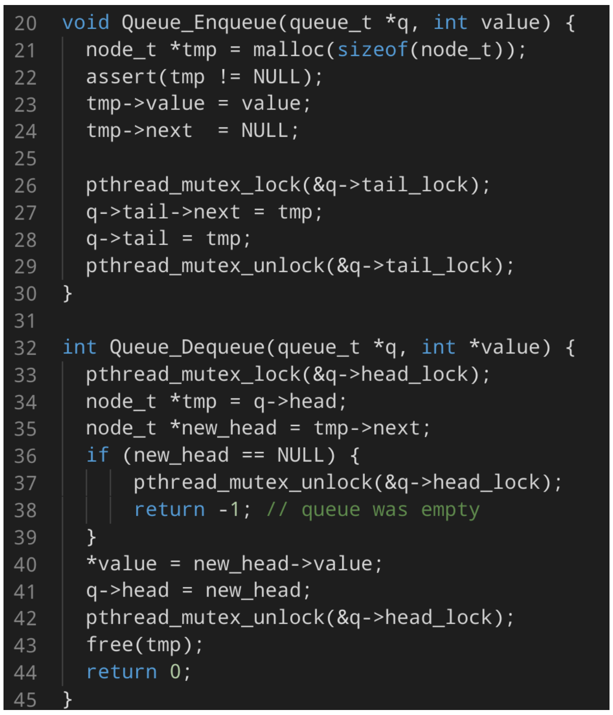
Concurrent Queues

It is always standard to add a big lock to entire structure.

Here is a more interesting concurrent queue designed by Michael and Scott.



```c
typedef struct __node_t {
  int value;
  struct __node_t *next;
} node_t;

typedef struct __queue_t {
  node_t *head;
  node_t *tail;
  pthread_mutex_t head_lock, tail_lock;
}

void Queue_Init(queue_t *q) {
  node_t *tmp = malloc(sizeof(node_t));
  tmp->next = NULL;
  q->head = q->tail = tmp;
  pthread_mutex_init(&q->head_lock, NULL);
  pthread_mutex_init(&q->tail_lock, NULL);
}
```



```c
void Queue_Enqueue(queue_t *q, int value) {
  node_t *tmp = malloc(sizeof(node_t));
  assert(tmp != NULL);
  tmp->value = value;
  tmp->next  = NULL;

  pthread_mutex_lock(&q->tail_lock);
  q->tail->next = tmp;
  q->tail = tmp;
  pthread_mutex_unlock(&q->tail_lock);
}

int Queue_Dequeue(queue_t *q, int *value) {
  pthread_mutex_lock(&q->head_lock);
  node_t *tmp = q->head;
  node_t *new_head = tmp->next;
  if (new_head == NULL) {
    pthread_mutex_unlock(&q->head_lock);
    return -1; // queue was empty
  }
  *value = new_head->value;
  q->head = new_head;
  pthread_mutex_unlock(&q->head_lock);
  free(tmp);
  return 0;
}
```