OPERATING SYSTEM

Study Notes

Zhijie Xia

Study Notes

github.com/zhijie-os

Contents

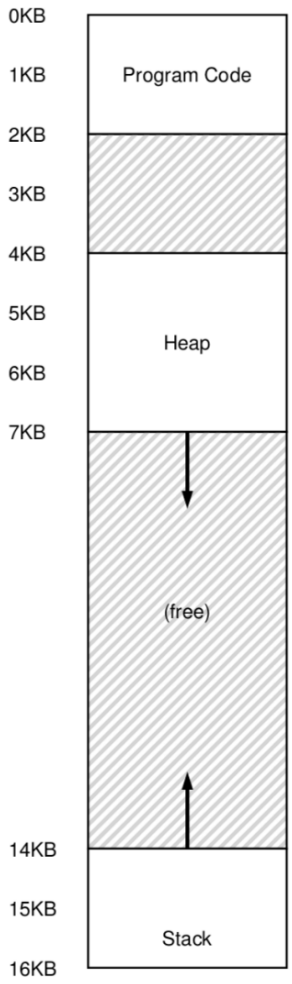
Chapter 1

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.

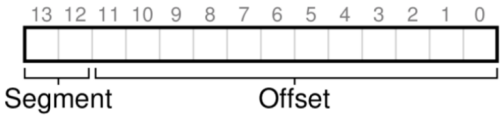
$$\text{Physical Address} = \text{Base Address} + \text{Offset}$$

Bound register is used to check boundary. If offset is greater than bound register value ⇒ 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 + $0.1 \cdot$ 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

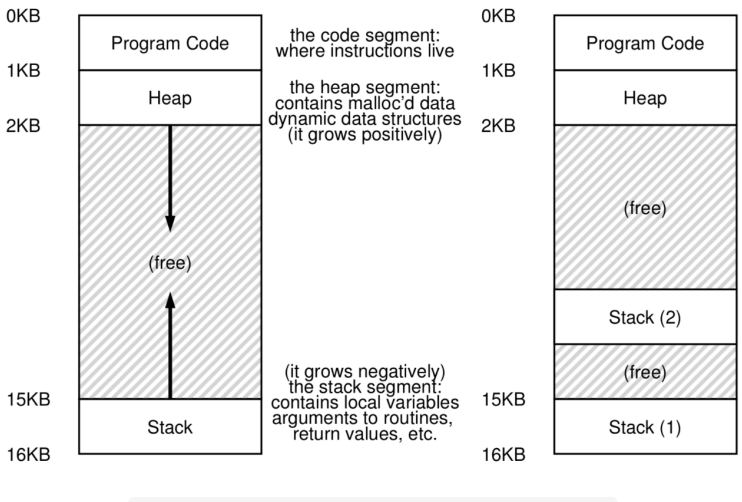
Thread: very much like process, except threads share the same address space and thus can access the same data.

2.1.1 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.

2.1.2 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.



2.1.3 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.

2.1.4 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.

2.1.5 Thread API

Lock

In POSIX library, a lock needs to be initialized

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

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

```
1  int pthread_mutex_lock(pthread_mutex_t *mutex);
2  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.

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

A thread must hold a lock to call either wait() or signal(). $pthread_cond_wait()$, puts the calling thread to sleep. $pthread_cond_signal()$ awakes the waiting thread.

The reason that $pthread_cond_wait()$ takes two parameter is because it needs to specify which thread to give the lock to. When $pthread_cond_wait()$ the calling thread release the lock and pass it to another thread.

2.1.6 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.

```
1 void lock() {
2   DisableInterrupts();
3 }
4 void unlock() {
5   EnableInterrupts();
6 }
```

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***

```c
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.

⇒ 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

```c
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 ⇒ 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.

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag  = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
```

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

```
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; //lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock
33                                     // (for next thread!)
34     m->guard = 0;
35 }
```

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.

2.1.7 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: A lock for the head, and a lock for the tail.

```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;
}
```

Concurrent Hash

A chaining hashtable is just an array of linked list. Since we have concurrent linked list, we could just use concurrent linked list to construct a hashtable. No more work needed.

```c
#define BUCKETS (101)

typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
    int i;
    for (i = 0; i < BUCKETS; i++)
        List_Init(&H->lists[i]);
}

int Hash_Insert(hash_t *H, int key) {
    return List_Insert(&H->lists[key % BUCKETS], key);
}

int Hash_Lookup(hash_t *H, int key) {
    return List_Lookup(&H->lists[key % BUCKETS], key);
}
```

2.1.8 Conditional Variables

There are many cases that we want to check whether a condition is true before continuing execute. For exmaple, a parent thread would like to wait its children to finish execution using $pthread_join()$.

A global conditional variable can used, and parent spins to wait the the global variable become true. However, we would like another approach that puts parent into sleep instead of spinning.

Definition and Routines

A ***condition variable*** is an explict ***queue*** that threads can put themselves on when some condition, which is not as desired by waiting on the condition.

From my understanding, it a thread A is waiting for a state, it would put it into the queue that every thread in the queues wants the same state. Thread B can make the state happen, and then thread B would wake up one or more those threads in the queue, including A.

Syntax and Usage

In pthread library, $pthreadcond_t$ ***name;*** can be used to create condition variable.

And a condition variable has two operations: ***wait()*** and ***signal***.

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

wait() is like "Hey, I need something and put me into sleep until that thing is avaiable."

signal() is like "Ok, I make the thing you wanted, and I am going to wake you up."

Note that wait() takes a lock as second parameter, and that is because it assumes the lock is hold by the caller when calling wait() and it would release the lock and be put into sleep (automically). The wait() won't return unless the lock is reacquired by the caller.

```
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
  Pthread_mutex_lock(&m);
  done = 1;
  Pthread_cond_signal(&c);
  Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
  printf("child\n");
  thr_exit();
  return NULL;
}

void thr_join() {
  Pthread_mutex_lock(&m);
  while (done == 0)
    Pthread_cond_wait(&c, &m);
  Pthread_mutex_unlock(&m);
}
```

But, why! Why! Why we need the state variable?

```
void thr_exit() {
  Pthread_mutex_lock(&m);
  Pthread_cond_signal(&c);
  Pthread_mutex_unlock(&m);
}

void thr_join() {
    Pthread_mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Imagine, child thread executes upon the creation. It would signal when the queue is empty and exit. When the control is returned to the parent, parent would still call wait() and no one is going to wake it up. A ***If*** instead a ***While*** would probabily makes much sense.

TIP: Always Hold the Lock While Signaling

Mandatory: Hold the Lock While Calling Wait.

Producer/Consumer (Bounded Buffer)

Problem Statement: Producers generate data items and place them in the buffer; consumers grab said items from the buffer and consume them in some way.

A real would example: Web server; A producer puts HTTP requests into a work queue. While, the consumer threads

```c
int buffer;
int count = 0; // initially, empty

void put(int value) {
  assert(count == 0);
  count = 1;
  buffer = value;
}

int get() {
  assert(count == 1);
  count = 0;
  return buffer;
}
```

We use int for simplicty. This buffer only has one slot for item. put() should set count to 1 when the slot is empty and get() should set count to 0 when the slot is filled.

A broken approach:

```c
int loops; // must initialize somewhere...
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++){
    Pthread_mutex_lock(&mutex);          // p1
    if (count == 1)                      // p2
      Pthread_cond_wait(&cond, &mutex); // p3
    put(i);                              // p4
    Pthread_cond_signal(&cond);          // p5
    Pthread_mutex_unlock(&mutex);        // p6
  }
}

void *consumer(void *arg) {
 int i;
 for(i = 0; i < loops; i++) {
   Pthread_mutex_lock(&mutex);          // c1
   if (count == 0)                      // c2
    Pthread_cond_wait(&cond, &mutex);  // c3
    int tmp = get();                     // c4
    Pthread_cond_signal(&cond);          // c5
    Pthread_mutex_unlock(&mutex);        // c6
    printf("%d\n", tmp);
  }
}
```

This works only when there is exactly one producer and one consumer, since the condition variable is set in a way that producer and consumer take turns. However, this is not useful anyway. Suppose there are two consumer c1,c2 and one producer p1.

Upon the creation of the c1, it runs and check the buffer is empty it waits. Then p1 runs and put item into the buffer and it call signal. Now c2 passed the if statement, and both c1,c2 call get().

You see here, the signal only tells at the moment the world is desired and you can wake up. But when

waiting one wakes up, it could find the work has already changed once again, i.e, not desired.

A "better" broken approach: use while instead of if

```
int loops; // must initialize somewhere...
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++){
    Pthread_mutex_lock(&mutex);          // p1
    if (count == 1)                      // p2
      Pthread_cond_wait(&cond, &mutex); // p3
    put(i);                              // p4
    Pthread_cond_signal(&cond);          // p5
    Pthread_mutex_unlock(&mutex);        // p6
  }
}

void *consumer(void *arg) {
 int i;
 for(i = 0; i < loops; i++) {
   Pthread_mutex_lock(&mutex);          // c1
   while (count == 0)                   // c2
    Pthread_cond_wait(&cond, &mutex);  // c3
    int tmp = get();                     // c4
    Pthread_cond_signal(&cond);          // c5
    Pthread_mutex_unlock(&mutex);        // c6
    printf("%d\n", tmp);
  }
}
```

This would address the issue in last approach, c1 and c2 can safely consume the product. When c1 wakes up, it checks whether the world is desired again (because of while).

This "always use while loops" is called ***Mesa semantics***.

However, the above approach is still broken due to the nondeterministic signaling.

1. both c1 and c2 sleep when they find there is item in the buffer.

2. p1 puts the item in the buffer and signal one consumer to wake up, and go to sleep

3. assume c1 wakes up, and cosumed the one item.

4. c1 signals on the condition variable,and went to sleep.

5. c2 wakes up, and check the buffer is empty, back to sleep.

6. all p1,c1,c2 are sleeping

This is due to there is one condition variable, and the signaling is nondeterministic. A consumer should never wakes up a consumer, and a producer should never wake up a producer.

Correct solution:

Producer wakes up consumer, and consumer wakes producer. Thus, two condition variables needed.

```
1  cond_t  empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5    int i;
6    for(i=0;i<loops;i++){
7      Pthread_mutex_lock(&mutex);
8      while (count == 1)
9        Pthread_cond_wait(&empty, &mutex);
10     put(i);
11     Pthread_cond_signal(&fill);
12     Pthread_mutex_unlock(&mutex);
13   }
14 }
15
16 void *consumer(void *arg) {
17   int i;
18   for (i=0;i<loops;i++){
19     Pthread_mutex_lock(&mutex);
20     while (count == 0)
21       Pthread_cond_wait(&fill, &mutex);
22     int tmp = get();
23     Pthread_cond_signal(&empty);
24     Pthread_mutex_unlock(&mutex);
25     printf("%d\n", tmp);
26   }
27 }
28
```

Note that two condition variables are in using: ***empty*** and ***fill***, if is used to replace while in producer part.

When producer filled the buffer, it signals full "Hey, we have item in the buffer". And it waits for empty "Well, call me when the buffer is empty".

When consumer emptied the buffer, it signals empty "Hey, producer. The buffer is empty, please fill". And it waits for full "Also, call me when it is filled, I need more".

General solution:

We let producer produces multiple items when wakes up. And let consumer consumes multiple items when wakes up.

With single pair of producer and consumer, this approach is more efficient as it reduces context switches; with multiple prodcuers or consumers, it even allows concurrent producing or consuming to take place.

```c
int buffer[MAX];
int fill_ptr = 0;
int use_ptr  = 0;
int count    = 0;

void put (int) value) {
  buffer[fill_ptr] = value;
  fill_ptr = (fill_ptr + 1) % MAX;
  count++;
}

int get() {
  int tmp = buffer[use_ptr];
  use_ptr = (use_ptr + 1) % MAX;
  count--;
  return tmp;
}
```

```c
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++){
    Pthread_mutex_lock(&mutex);            //p1
    while (count == MAX)                   //p2
      Pthread_cond_wait(&empty, &mutex);   //p3
    put(i);                                //p4
    Pthread_cond_signal(&fill);            //p5
    Pthread_mutex_unlock(&mutex);          //p6
  }
}

void *consumer(void *arg) {
  int i;
  for (i = 0; i<loops;i++){
    Pthread_mutex_lock(&mutex);            //c1
    while (count == 0)                     //c2
      Pthread_cond_wait(&fill, &mutex);    //c3
    int tmp = get();                       //c4
    Pthread_cond_signal(&empty);           //c5
    Pthread_mutex_unlock(&mutex);          //c6
    printf("%d\n", tmp);
  }
}

```

Covering Conditions: Broadcast

```c
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t c;
mutex_t m;

void * allocate(int size) {
  Pthread_mutex_lock(&m);
  while (bytesLeft < size)
    Pthread_cond_wait(&c, &m);
  void *ptr = ...; // get mem from heap
  bytesLeft -= size;
  Pthread_mutex_unlock(&m);
  return ptr;
}

void free(void *ptr, int size) {
  Pthread_mutex_lock(&m);
  bytesLeft += size;
  Pthread_cond_signal(&c); // whom to signal??
  Pthread_mutex_unlock(&m);
}
```

Problem Scenario:

There are two threads Ta,Tb. Ta asks for 10 bytes and Tb asks for 100 bytes. At the creations of Ta and Tb, there is 0 bytes free memory. Therefore Ta and Tb sleep. When Tf frees up 50 bytes, Tf signaled Tb which needs 100 bytes. Tb would sleep again, and no memory would be allocated even Ta's request can be fulfilled.

Solution:

Instead of $pthread_cond_signal()$, we can use $pthread_cond_broadcast$.

However, the cost is very noticable since all threads would be waken up.

2.1.9 Semaphore

Definition:

A semaphore is an object with an integer value that we can manipulate with two routines; in POSIX, these routines are $sem_wait()$ and sem_post.

The initial value of the semaphore determines its behavior, it is necessary to initialize the semaphore to some value.

```
1  #include <semaphore.h>
2  sem_t s;
3  sem_init(&s, 0, 1);
```

After initialize the semaphore, one can interact with it using $sem_wait()$ and sem_post.

```
1  int sem_wait(sem_t *s) {
2      decrement the value of semaphore s by one
3      wait if value of semaphore s is negative
4  }
5
6  int sem_post(sem_t *s) {
7      increment the value of semaphore s by one
8      if there are one or more threads waiting, wake one
9  }
10
```

The value of the semaphore is equal to the number of, when negative,waiting threads.
And the initial value is the number of resources that can be give away at start.

Binary Semaphore (Locks)

```
1  sem_t m;
2  sem_init(&m, 0, X); // i
3
4  sem_wait(&m);
5  // critical section here
6  sem_post(&m);
```

This is essential just a lock, when semaphore is initialized to 1.
However, this is useless since the lock already serves the purpose.

Producer/Consumer (Bound Buffer):Semaphore approach

Note that semaphore works like a combination of lock and condition variable.

```
1  int buffer[MAX];
2  int fill = 0;
3  int use = 0;
4
5  void put(int value) {
6    buffer[fill] = value; //Line F1
7    fill = (fill + 1) % MAX; //Line F2
8  }
9
10 int get() {
11   int tmp = buffer[use]; // Line G1
12   use = ( use + 1) % MAX; //Line G2
13   return tmp;
14 }
15
```

```c
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);  //Line P1
        put(i);            //Line P2
        sem_post(&full);   //Line P3
    }
}

void *consumer(void *arg) {
    int i,tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);      //Line C1
        tmp = get();          //Line C2
        sem_post(&empty);     //Line C3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX are empty
    sem_init(&full, 0, 0);    // 0 are full
    // ...
}
```

However, there would be a race condition at the put() and get():

1. Pa gets to run get() first, it first puts item in slot i.
2. just before Pa increment the slot counter, it gets interrupted.
3. Pb still have slot counter at slot i, and Pb overwrites slot i.

That is , there is no mutual exclusion for put() and get(), but only signaling whether one can put or get.

```c
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex); // Line P0 (NEW LINE)
        sem_wait(&empty); // Line P1
        put(i);           // Line P2
        sem_post(&full);  // Line P3
        sem_post(&mutex); // Line P4 (NEW LINE)
    }
}
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex); // Line C0 (NEW LINE)
        sem_wait(&full);  // Line C1
        int tmp = get();  // Line C2
        sem_post(&empty);  // Line C3
        sem_post(&mutex); // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

Now, the revised solution use semaphore (like a lock) to add mutual exclusion.
However, it still doesn't work because deadlock since semaphore don't increament itself when sleep
The consumers and producers share a single lock.

1. Consumer C1 runs first, the use sem-wait(&mutex) to required the lock, mutex = 0.
2. C1 continue, and calls sem-wait(&full), however there is no item in the slot. And C1 sleeps.
3. When any other producers or consumers try to do something, they don't have the lock, any sem-wait($mutex) would decrease the mutex to negative value and sleeps forever.

```c
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty); // Line P1
        sem_wait(&mutex); // Line P1.5 (MUTEX HERE)
        put(i);           // Line P2
        sem_post(&mutex); // Line P2.5
        sem_post(&full);  // Line P3
    }
}
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);  // Line C1
        sem_wait(&mutex); // Line C1.5 (MUTEX HERE)
        int tmp = get();  // Line C2
        sem_post(&mutex); // Line C2.5 (AND HERE)
        sem_post(&empty); // Line C3
        printf("%d\n", tmp);
    }
}
```

Now a truly working vision,

To resolve the deadlock, simply reduce the scope of lock at this case.

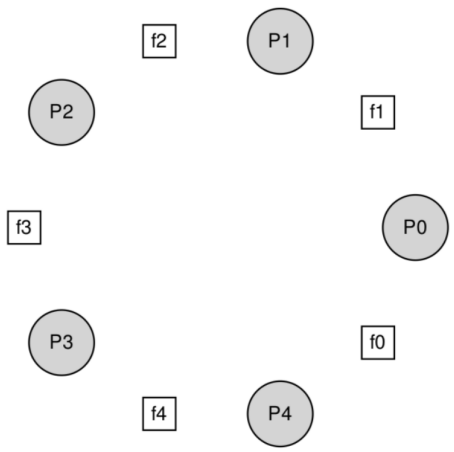
Reader-Writer Locks

For concurrent list operations, we need to guarantee that no two insertion races, but it is ok for multiple readers to lookup concurrently.

reader-write lock: The implementation is in reader-writer.cpp

Dining Philosophers

Problem statement: There are five "philosophers" sitting around a table. Between each pair of philosophers is a single fork (and thus, five total). The philosophers each have times where they think and don't need any forks, and times where they eat. In order to eat, a philosopher needs two forks, both the one on their left and the one on their right



Here is the behavior for a philosopher:

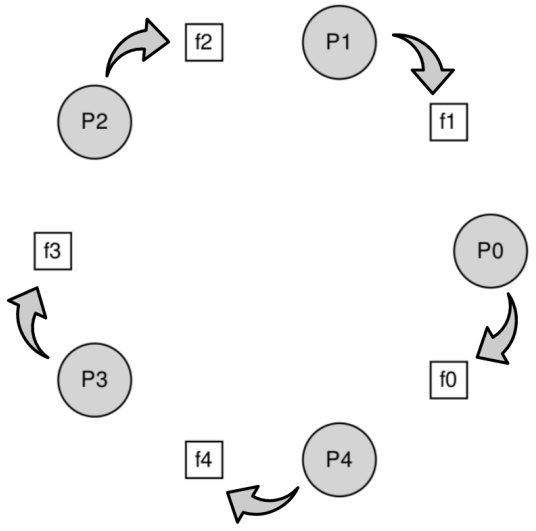
```
while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```

Some help functions needed

```
int left(int p){
    return p;
}

int right(int p){
    return (p+1)%5;
}
```

Broken solution:

```
// potientially deadlock: if every philosopher grabs the left one
// that is, all threads makes to sem_wait(&forks[left(p)]) and no further
void get_forks(int p){
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
}

void put_forks(int p){
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
}
```



Dijkstra solution: break the dependency
Make one philosopher behaves different than other ones.

```
void get_forks(int p){
    // letting philosopher 4 behaves differently, there would be deadlock
    // since there won't be a circle now.
    if(p==4){
        sem_wait(&forks[right(p)]);
```

```
        sem_wait(&forks[left(p)]);
    }else{
        sem_wait(&forks[left(p)]);
        sem_wait(&forks[right(p)]);
    }
}
```

Thread Throttling

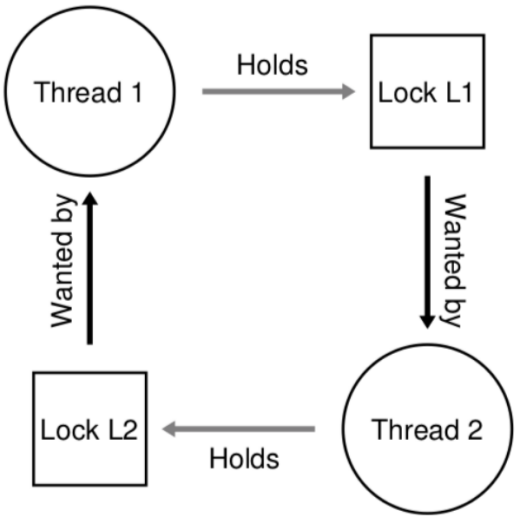
How can a programmer prevent "too many" threads from doing something at once and bogging the system down?

Answer is **Throttling:** decide a threshold, use semaphore to limit the number of threads concurrently executing.

How implement semaphore

```
typedef struct Semaphore{
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
}semaphore;

void sem_init(sem *s, int value){
    s-> value = value;
    cond_init(&s->cond);
    mutex_init(&s->lock);
}

void sem_wait(sem *s){
    pthread_mutex_lock(&s->lock);

    // however, this value would never go negative
    // easy to implement, linux's approach
    while(s->value <=0){
        pthread_cond_wait(&s->cond,&s->lock);
    }
    s->value = s->value - 1;
    pthread_mutex_unlock(&s->lock);
}

void sem_post(sem *s){
    pthread_mutex_lock(&s->lock);
    s->value = s->value - 1;
    pthread_cond_signal(&s-cond);
    pthread_mutex_unlock(&s->lock);
}
```

2.1.10 Deadlock bugs



Four condtions need to be hold for a deadlock to occur:

1. **Mutual exclusion:** Threads claim exclusive control of resources.
2. **Hold-and-wait:** Threads hold the resource and wait for other necessary resource, wont let go.
3. **No preemption:** Resources cannot be forcibly removed from the threads that are holding them.
4. **Circular wait:** There exists a circular chain of threads such that each threads holds one or more resources that are being requested by the next thread in the chain.

Prevention

Make one of the four condition false all the time to prevent deadlock. However, this is not practical.

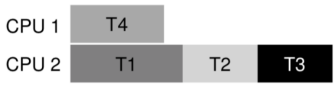
Deadlock avoidance via Scheduling

Instead of deadlock prevention, in some scenarios deadlock avoidance is preferable. Avoidance requires some global knowledge of which locks various threads might grab during their execution, and subsequently, schedules said threads in a way as to guarantee no deadlock can occur.

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

From the images, it is possible to schedule threads to avoid deadlock
if we need the resources that eac thread wants before hand.

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no



Also, the scheduler could procude length-considerable scheduling.

Thus, this approach can be only used in limited environment, for example, in an embedded system where one has full knowledge of the entire set of tasks must be run and the locks that they need.

Also, such approach can limit the degree of concurrency.

Detection And Recovery

One strategy can allow deadlock to occur occasionally, and then take some action once it has been detected.

A deadlock detector runs periodically, building a resource graph and checking it for cycles.

2.1.11 Event-Based Concurrency

Why event-based concurrency:

1. Managing concurrency correctly in multi-threaded applications are challenging.
2. Multi-threaded application, the developer has little or no control over what is scheduled at a given moment in time.

Given the difficulty of building a general-purpose scheduler that works well in all cases for all workloads, one cannot expect the scheduler schedule optimally everytime. Thus, we have **event-based concurrency** to address the problem.

The basic idea: An Event Loop

One can simply wait for something to occur; when it does, you check what type of event it is and do the small amount of work it requires, which may include issuing I/O requests, or scheduling other events for future handling.

An example can be event-based server. Such applications are based around on an **event loop**.

```
while(1){
    events = getEvents();
    for( each event in events){
        processEvent(e); // event handler
    }
}
```

When event handler processes an event, it is the only work being done. It is just like a scheduler schedule the event. Thus, we can decide which event to handler next to take over control on which work should be scheduled next.

But how? How can the event-based server determines which events are taking place?

Important API: select() or poll()

In most systems, an API either called select() or poll() system calls to receive events.

The API enables a program to check whether there is any incoming I/O that should be attended to.

select()

```
int select( int nfds,
            fd_set *restrict readfds,
            fd_set *restrict writefds,
            fd_set *restrict errorfds,
            struct timeval *restrict timeout);
```

select() lets you check whether descriptors can be read from or written to. The former lets a server determine that a new packet has arrived. The latter lets the service know when to reply(i.e, the outbound queue is not full).

Also, there is "timeout" interval. Common usage is to set it to NULL, so that select() would be blocked indefinitely, until a descriptor is ready; another common usage is to set it to 0, so that select() return immediately.

Blocking vs Non-Blocking interfaces

Blocking(synchronous) interfaces do all the work before returning to the caller; non-blocking(asynchronous) interfaces begin some work but return immediately, and let the remaining work done in background.

Non-blocking interfaces are essential in the event-based approach.

Problem: Blocking System Calls

There is one issue with event-based programming: issue blocking system call.

Without the nature of overlapping I/O and other computations that multi-threaded programming has, event-based systems can only block until the call completes, i.e, no solution for the problem.

Thus, we make a rule for event-based systems : no blocking calls are allowed.

Asynchronous I/O

To overcome the issue of blocking calls in event-based system, modern operating system have introduced new ways to issue I/O requests to the disk system, referred to generically as asynchronous I/O.

The interfaces allow issuing I/O request and return immediately. Additional interfaces are also avaiable for the system to decide whether an I/O request is fulfilled.

```
struct aiocb { // AIO control block
          int             aio_fildes;     /* File descriptor */
          off_t           aio_offset;     /* File offset */
          volatile void  *aio_buf;        /* Location of buffer */
          size_t          aio_nbytes;     /* Length of transfer */
          int             aio_reqprio;    /* Request priority offset */
          struct sigevent aio_sigevent;   /* Signal number and value */
          int             aio_lio_opcode;/* Operation to be performed */
      };

int aio_read(struct aiocb *aiocbp);

// check whether the request referred to by aiocbp has completed.
// If yes, 0 is returned; otherwise, EINPROGRESS is returned.
int aio_error(const struct aiocb *aiocbp);
```

Problem: State Management

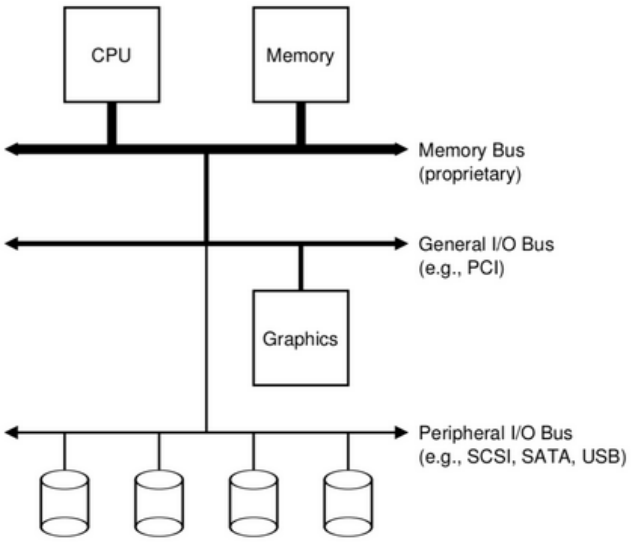
Another problem with event-based approach is that such code is commonly more complicated to write. For example, when an event handler issues an asynchronous I/O, it must package up some program state for the next event handler to use when the I/O finally completes.

Chapter 3

Persistence

3.1 I/O devices

Prototypical System Architecture

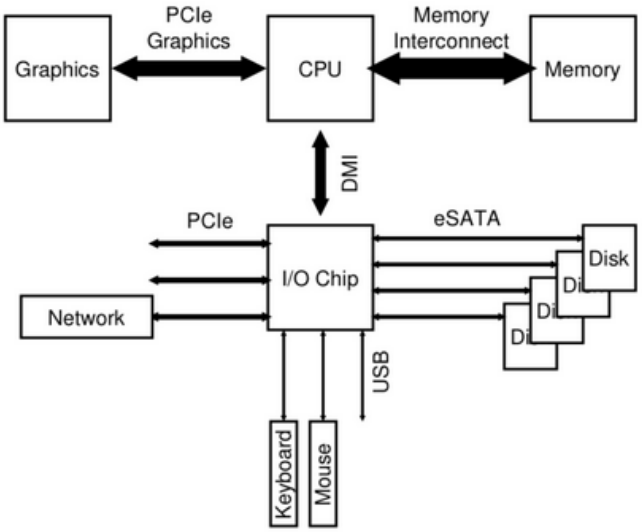


CPU and Memory are connected to the **Memory Bus**. Other devices like graphic card are connected to the system via a general **I/O bus**, which should be PCI(or its derivatives). There is even lower bus called **peripheral bus**, such that **SCSI,SATA** or **USB** can be attached here. Those connect slow devices to the system, including disks, mice and keyboards.

The idea is that: components that demand high performance are nearer the CPU. Lower performance components are further away.

Modern System Architecture

Modern systems use specialized chipsets and faster point-to-point interconnects to improve performance.


Graphics
PCIe
Graphics
CPU
Memory
Interconnect
Memory
DMI
PCIe
I/O Chip
eSATA
Disk
Network
USB
Keyboard
Mouse