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.

```
0KB   ┌──────────────┐
      │              │
1KB   │ Program Code │
      │              │
2KB   ├──────────────┤
      │////////////////│
3KB   │////////////////│
      │////////////////│
4KB   ├──────────────┤
      │              │
5KB   │              │
      │     Heap     │
6KB   │              │
      │              │
7KB   ├──────────────┤
      │//////│//////│
      │//////▼//////│
      │////////////////│
      │////////////////│
      │     (free)    │
      │////////////////│
      │////////////////│
      │//////▲//////│
      │//////│//////│
14KB  ├──────────────┤
      │              │
15KB  │     Stack    │
      │              │
16KB  └──────────────┘
```
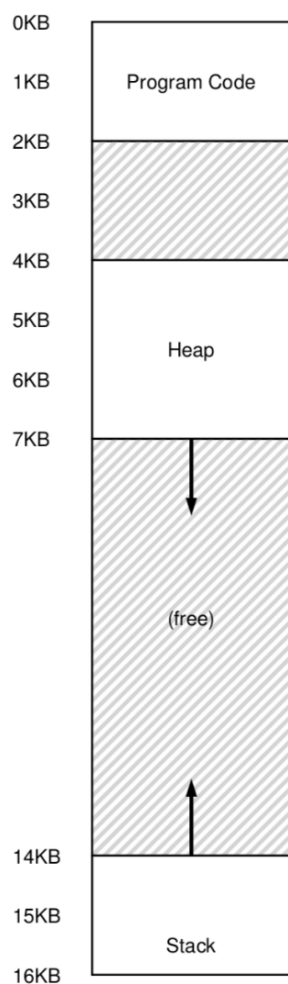
In canonical address space, we have three logically-different segments:

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 absolute 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

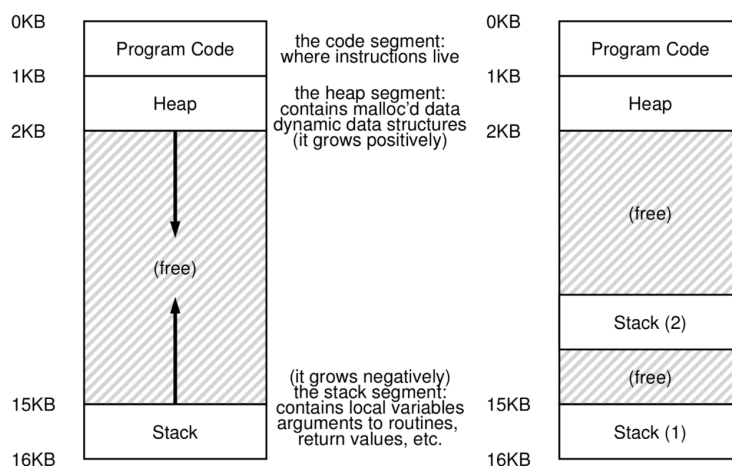Thread: very much like process, except threads share the same address space and thus can access the same data.

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

2.1.2 Benefit of using threads

11

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.

```
11  void *mythread(void *arg) {
12      char *letter = arg;
13      int i; // stack (private per thread)
14      printf("%s: begin [addr of i: %p]\n", letter, &i);//threads would share the same data
15      for (i = 0; i < max; i++) {
16          // at here, it would like
17
18          // ldr x1,counter
19          // add x1,x1,1   ---- if the context switch happens here, it could cause problem.
20          // str x1,counter
21          counter = counter + 1; // shared: only one
22      }
23      printf("%s: done\n", letter);
24      return NULL;
25  }
26
```

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

```
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.

2.1.5 Locks