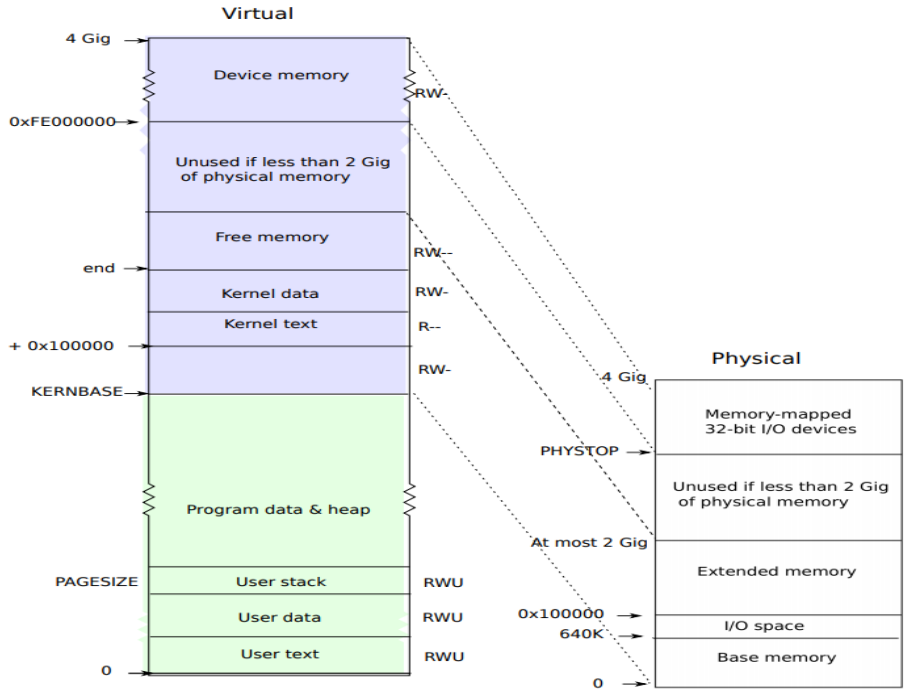
20CS2103A - Test-2 – Set2 - Key

1. **Give examples of why processes might wish to share parts of their address spaces. What kind of memory management hardware would support this sharing? How can memory protection be enforced when sharing is allowed?**

Ans: Code sharing (e.g. of utilities or libraries) or read-only data sharing is transparent to the processes concerned. It allows the system to economise on the use of physical memory by avoiding multiple copies. Segmentation hardware (or paging hardware with segmentation managed by the OS) is needed with write protection on the shared code or data. Sharing of writeable data areas allows processes to co-operate efficiently. They may wish to share only a portion of their address spaces. Fine grained segmentation hardware (many segments per process) would support this. If only paging hardware is available the OS may allow the processes to declare shareable regions of their data spaces and set up the shared pages accordingly.

2. **Compare and contrast xv6 virtual memory system with Linux Virtual Memory System.**

Ans:



A process's user memory starts at virtual address zero and can grow up to KERNBASE, allowing a process to address up to 2 gigabytes of memory. When a process asks xv6 for more memory :

- finds free physical pages to provide the storage
- adds PTEs to the process's page table that point to the new physical pages
- sets the PTE_U, PTE_W, and PTE_P flags in these PTEs.
- xv6 leaves PTE_P clear in unused PTEs.
- Different processes' page tables translate user addresses to different pages of physical memory, so that each process has private user memory
- Xv6 includes all mappings needed for the kernel to run in every process's page table; these mappings all appear above KERNBASE.
- It maps virtual addresses KERNBASE:KERNBASE+PHYSTOP to 0:PHYSTOP.

3. **The UNIX operating system uses the copy-on-write technique in order for the parent and child processes to share memory until the point where one of them tries to modify a page frame of the shared memory. Just before the modification takes place, the system makes a copy of the frame and allocates it to the child process. State the similarities and the differences of this technique as compared to thread methodology.**

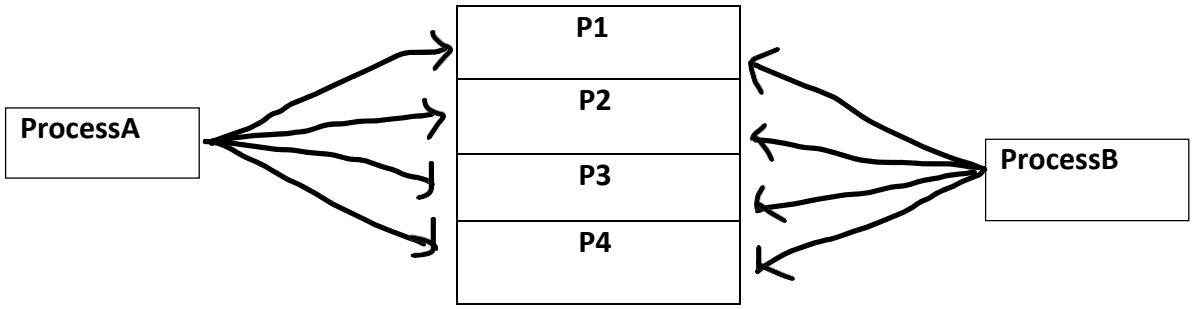
Ans:

Copy on write is resource management technique. One of it's main use is it's use in fork() system call.

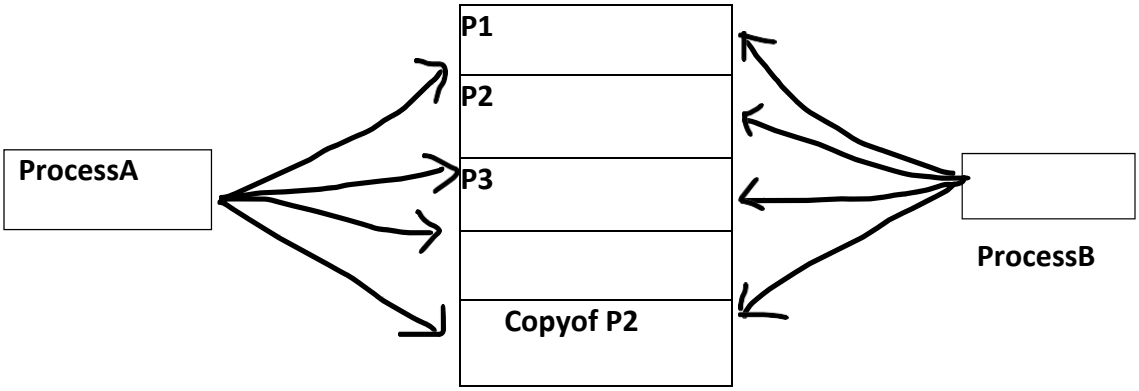
The idea behind copy on write is that when parent process creates a child process both these process initially will share same pages of memory and shared pages will be marked copy-on-write which means of any of

process will try to modify shared pages then only a copy will be created and modifications will be done on copy and not affect the original.

Before write:



After Write:



Drawbacks:

Any changes made to original volume are performed have double write penalty. A Write to a volume having copy-on-write snapshot takes 1 read and 2 writes.

4. **In xv6, briefly explain Assembly trap handlers and C trap handlers. Illustrate Assembly bootstrap and C bootstrap.**

Ans:

Cpu execute user progs when certain events happen User prog

traps to kernel program

- System calls (request or service)
- Interrupts (external device attention needed)
- Program fault (illegal action by os)

When the above event happen cpu execute the special "int" instruction which cause jump between os code for hardware interrupts device sends signal to os and cpu runs in. Trap instruction has parameter (int n) indicating type of interrupt. This cause cpu to run "int" instruction

Before trap: eip pointing to user program, Esp

pointing to stack

If trap occurs cpu runs (int n)

1. Cpu has IDT(interrupt descriptor table) and fetches n^th entry
2. Save esp to internal register
3. Switch esp to kernel stack
4. On kernel stack save old esp,eip
5. Load new eip from IDT

Result: Ready to run kernel Trap Handler code.

(i) Stack and few other registers also stored

(ii) Cpu privilege steps are performed on IDT access

(iii) If interupt occurs while handing interupt no need to save points again

Why do we need seperate trap instruction?

(i) Cpu executes user code in low privilege level, but os code has high privilege.

(ii) User program sometimes cannot run kernel code properly

(iii) someone needs to change privilege level and control to kernel.

(iv) All of these are dow when cpu executing "int n"

As part of handling trap we have to save state to kernel stack , to return to old program i.e. user program.

This context is stored in trapframe (struct trapframe) along with same extra info like trapno. This trapframe is pushed to kernel stack and esp points to top of kernel stack

kernel trap handler

IDT is set up in table points to kernel trap handler

all traps push more registers to kernel stack like builder trap frame

%ds %es %fs %gs pushal->general purpose register Builder trap

name

All registers in trapframe are pushed by kernel alltraps assembly code or function

By noco we have kernel stack and populated stack frame %esp points to top of kernel stack

After all the basic work is done assembly code call trap function(c function) with an argument as trapframe □%esp. so we push %esp to stack.

we push argument to stack before calling function void

trapframe(struct trapframe *tf);

Based on trapno the code does various things

if trapno == T_SYSCALL, then it calls the system call code

Traphandler invokes syscall function

It looks for number in eax and alls corresponding function and returns value of syscall in eax And based on value that particular syscall is invoked

whenever trap occurs □ "int n" index □ trap function which calls syscall □ looks in %eax

It is a device interrupt say keyboard then corresponding kbdintr() is called ideintr() is disk, times then respective code is runned.

Timer is a special hardware interrupt generated to trap to kernel, on timer interrupt process "yields" to cpu.

Called on Timeslice.

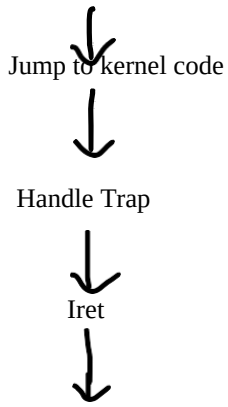
Yield changes state of process to RUNNABLE and give up cpu.

To return from Trap:

Trapret is used to reverse i.e., undo (or) popoll flu contents of kernel stack into corresponding register iret will decrement privilege and undo all changes and return to

"pre -trap" code.

Uprog "int n"



11) **privilege under x86:**

(i) x86 cpus have multiple privilege levels

-four "rings"

-Ring 0 has highest privilege -os

-Ring 3 has lowest -user

(ii) 2 types of instructions

-privileged -does some sensitive operation (writing to cr3 reg)

-unprivileged

(iii) when user programs require os services, CPU moves to higher privilege levels and execute os code that contains privileged instructions

To call a function:

push function args to stack call function

Allocate local variables to stack run function code

return

register values should be saved

Callersavesomeregisterandcallersavesomeregisters. Localvar, argumentsstoredinstack

caller expects the registers to have some value on return register value stored in eax by called

5a.

List x86 protection modes and briefly explain x86 interrupt hardware. Illustrate xv6 kernel stack before calling trap X86 protection

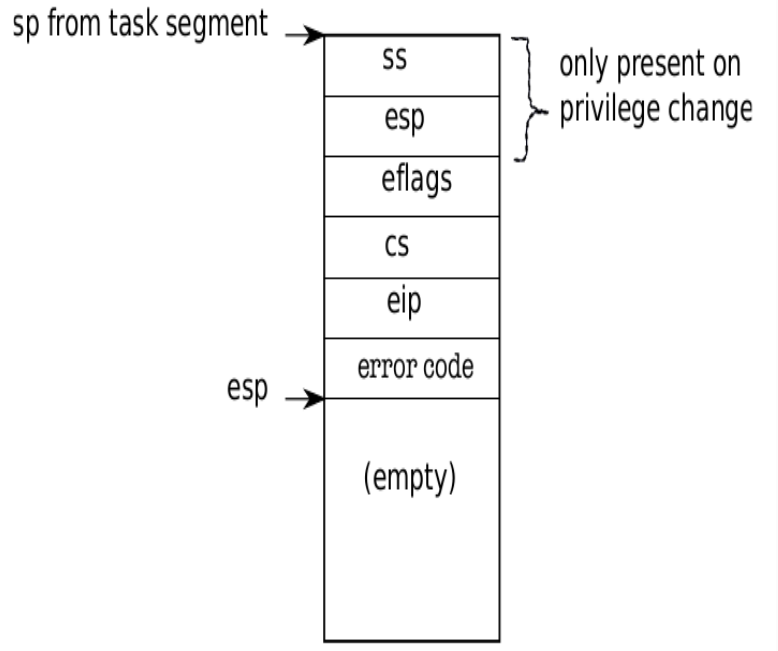
The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege). In practice, most operating systems use only 2 levels: 0 and 3, which are then called kernel mode and user mode, respectively. The current privilege level with which thex86 executes instructions is stored in %cs register, in the field CPL.

On the x86, interrupt handlers are defined in the interrupt descriptor table (IDT). The IDT has 256 entries, each giving the %cs and %eip to be used when handling the corresponding interrupt.

To make a system call on the x86, a program invokes the int n instruction, where n specifies the index into the IDT. The int instruction performs the following steps:

- Fetch the n'th descriptor from the IDT, where n is the argument of int.
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
- Save %esp and %ss in a CPU-internal registers, but only if the target segment selector's PL < CPL.
- Load %ss and %esp from a task segment descriptor.
- Push %ss.
- Push %esp.
- Push %eflags.
- Push %cs.
- Push %eip.
- Clear some bits of %eflags.
- Set %cs and %eip to the values in the descriptor.

The int instruction is a complex instruction, and one might wonder whether all these actions are necessary. The check CPL <= DPL allows the kernel to forbid systems for some privilege levels. For example, for a user program to execute int instruction succesfully, the DPL must be 3. If the user program doesn't have the appropriate privilege, then int instruction will result in int 13, which is a general protection fault. As another example, the int instruction cannot use the user stack to save values, because the user might not have set up an appropriate stack so that hardware uses the stack specified in the task segments, which is setup in kernel mode.



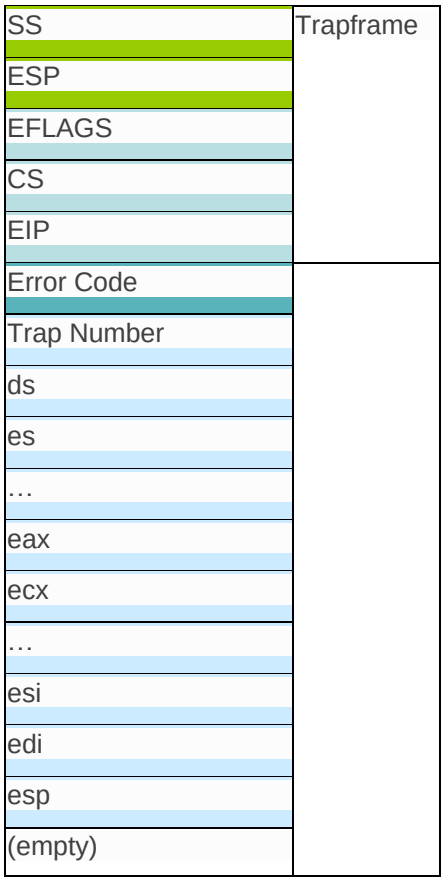


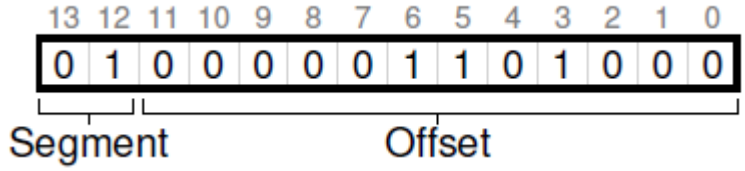
Figure. Kernel stack after an int instruction.

Figure shows the stack after an int instruction completes and there was a privilege-level change (the privilege level in the descriptor is lower than CPL). If the int instruction didn't require a privilege-level change, the x86 won't save %ss and%esp. After both cases, %eip is pointing to the address specified in the descriptor table, and the instruction

at that address is the next instruction to be executed and the first instruction of the handler for int n. It is job of the operating system to implement these handlers, and below we will see what xv6 does. An operating system can use the iret instruction to return from an int instruction. It pops the saved values during the int instruction from the stack, and resumes execution at the saved %eip.

5b. For a given virtual address of a segmented system in binary number, write and explain segmentation control flow algorithm that uses segment registers.

In the given example, then, if the top two bits are 00, the hardware knows the virtual address is in the code segment, and thus uses the code base and bounds pair to relocate the address to the correct physical location. If the top two bits are 01, the hardware knows the address is in the heap, and thus uses the heap base and bounds. Let's take our example heap virtual address from above (4200) and translate it, just to make sure this is clear. The virtual address 4200, in binary form, can be seen here:



As you can see from the picture, the top two bits (01) tell the hardware which segment we are referring to. The bottom 12 bits are the offset into the segment: 0000 0110 1000, or hex 0x068, or 104 in decimal. Thus, the hardware simply takes the first two bits to determine which segment register to use, and then takes the next 12 bits as the offset into the segment.

By adding the base register to the offset, the hardware arrives at the final physical address. Note the offset eases the bounds check too: we can simply check if the offset is less than the bounds; if not, the address is illegal. Thus, if base and bounds were arrays (with one entry per segment), the hardware would be doing something like this to obtain the desired physical address:

```
1  // get top 2 bits of 14-bit VA
2  Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3  // now get offset
4  Offset  = VirtualAddress & OFFSET_MASK
5  if (Offset >= Bounds[Segment])
6      RaiseException(PROTECTION_FAULT)
7  else
8      PhysAddr = Base[Segment] + Offset
9      Register = AccessMemory(PhysAddr)
```

Segmentation control flow algorithm that uses segment registers.

In our running example, we can fill in values for the constants above. Specifically, SEG MASK would be set to 0x3000, SEG SHIFT to 12, and OFFSET MASK to 0xFFF.

6a(i) Consider a machine with 32 KB physical memory and a 16-bit virtual address space. If the page size is 4KB and PageTable Entry (PTE) is 4 bytes, what is the size of the page table?

Number of entries in page table:
= (virtual address space size) / (page size)
= $2^{16}/2^{12} = 2^{4} = 16$
= Number of pages = 16
Size of page table = (number of entries in page table)*(size of PTE)
= 16 * 4KB = 64KB
(ii)

6b(ii) A computer system uses non-segmented 128-bit virtual addresses mapping pages of size 8 Kbytes with 256 Mbytes of physical memory. Show why a multi-level page table would not be effective here.

The number of levels in the page table must remain moderate in order for the cost of a lookup operation to be acceptable.

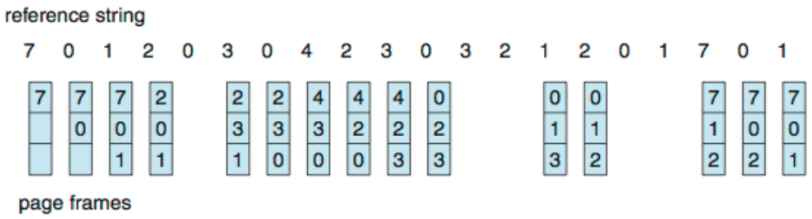
In this case, if each level mapped 10 bits then twelve levels would be required and twelve memory accesses would be needed for every translation.

In contrast, if fewer levels were to be used, then each page table would become larger.

This would only be acceptable if memory was used in a small number of contiguous segments.

6B. Assume that memory contains only three frames. Initially all three frames are empty. The page reference string is 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1. Find how many page faults occurs using FIFO, Optimal Page replacement, and Least Recently Used algorithms. Pictorially show which pages are replaced.

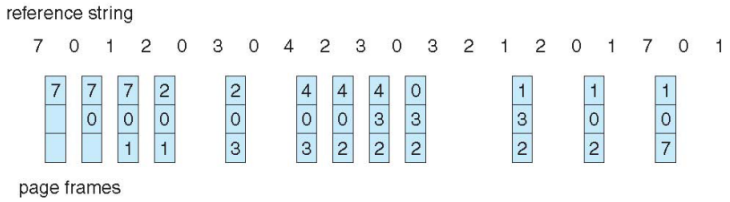
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

Least Recently Used (LRU) Algorithm

Use past knowledge rather than future
Replace page that has not been used in the most amount of time
Associate time of last use with each page



12 faults – better than FIFO but worse than Belady's/ OPT

Optimal algorithm: In this algorithm, we replace the page frame that **will not be used for the longest period of time in the near future** in the given reference string.

The page frame table for optimal replacement algorithm is:

7	**0**	**1**	**2**	**0**	**3**	**0**	**4**	**2**	**3**	**0**	**3**	**2**	**1**	**2**	**0**	**1**	**7**	**0**	**1**
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
F	**F**	**F**	**F**		**F**		**F**			**F**			**F**				**F**		

After (7,0,1), page frame **2** will replace page frame **7** since page frame 7 is going to be referenced only one time in the future.

This page replacement is having 9-page **faults**.

7. Many systems have a swap-atomic instruction that swaps the value of a register with that of a memory location. Show how such an instruction may be used to implement an atomic test-and-set. (4.5 M)

Ans: Implementing test set

An instruction that exchanges a value in memory with a value in a register has been used on a variety of machines. The primary purpose for these swap instructions is to provide an atomic operation for reading from and writing to memory, which has been used to construct mutual-exclusion mechanisms in software for process

synchronization. In fact, there are other forms of hardware instructions that have been used to support mutual exclusion, which include the classic test-and-set instruction.

An atomic test-and-set operation usually acts on a single bit in memory. It tests the bit, sets it to one, and returns its old value. Thus at the completion of the operation the value of the bit is one (locked), and the return value indicates whether it was already set to one prior to this operation. The operation is guaranteed to be atomic, so if two threads on two processors both issue the same instruction on the same bit, one operation will complete before the other starts. Further, the operation is also atomic with respect to interrupts, so that an interrupt can occur only after the operation completes. Such a primitive is ideally suited for simple locks. If the test-and-set returns one, the calling thread owns the resource. If it returns zero, the resource is locked by another thread. Unlocking the resource is done by simply setting the bit to zero. Some examples of test-and-set instructions are BBSS I (Branch on Bit Set and Set Interlocked) on the V AX-11 [Digi 87] and LDSTUB (LoaD and STore Unsigned Byte) on the SPARC

A swap instruction exchanges a value in memory with a value in a register. This is shown below, which depicts a load instruction, a store instruction, and a swap instruction using an RTL (register transfer list) notation.

Load Instruction

r[2] = M[x];

Store Instruction

M[x] = r[2];

Swap Instruction

r[2] = M[x]; M[x] = r[2];

8. How is a Thread Create? Why is communication and data sharing among threads faster than processes? Illustrate the reasons (4.5 M)

Ans: Thread creation: 2M

```
#include <pthread.h>
int pthread_create(     pthread_t*     thread,
                        const pthread_attr_t* attr,
                        void*          (*start_routine)(void*),
                        void*          arg);
```

- thread: Used to interact with this thread.
- attr: Used to specify any attributes this thread might have.
  - Stack size, Scheduling priority, …
- start_routine: the function this thread start running in.
- arg: the argument to be passed to the function (start routine)
  - *a void pointer* allows us to pass in *any type of* argument.

Why is communication and data sharing among threads faster than processes? Illustrate the reasons (2.5M)

Communication and data sharing among threads faster than process because very little memory copying is required (just the thread stack), threads are faster to start than processes. Not having separate copies means that different threads can read and modify a shared pool of memory easily. Threads of the same process share memory with the process they belong to. Context switching between processes is more expensive

9. Maintenance of the stack frame and the entities included inside it (local variables, return address, etc.) is achieved with the help of base pointer/frame pointer (EBP), stack pointer (ESP), instruction pointer (EIP). With Run-Time Stack Usage in 32-Bit GCC, Illustrate how a function call is made and the various steps involved during the process. main( ) { int x = 10;int y =20;int z; z = add(10, 20);z++; }

Function Call in C :

we consider the run-time behavior of a.out during execution. The run-time behavior of a program stems mainly from function calls. The following discussions apply to running C programs on 32-bit Intel x86 processors. On these machines, the C compiler generated code passes parameters on the stack in function calls. During execution, it uses a special CPU register (ebp) to point at the stack frame of the current executing function.

Run-Time Stack Usage in 32-Bit GCC

Consider the following C program, which consists of a main() function shown on the left-hand side, which calls a sub() function shown on the right-hand side.

```
-------------------------------------------------------------
main()                          | int add(int a, int b)
{                               | {
int x, y, z;                    | int u, v;
x = 10; y = 20;                 | u = 0; v = 0;
z = add(x, y);                  | return a+b+u+v;
printf("c=%d\n", z);            | }
}
-------------------------------------------------------------
```

(1) When executing a.out, a process image is created in memory, which looks (logically) like the diagram shown in Fig., where Data includes both initialized data and bss.
(2) Every CPU has the following registers or equivalent, where the entries in parentheses denote registers of the x86 CPU:
PC (IP): point to next instruction to be executed by the CPU.
SP (SP): point to top of stack.
FP (BP): point to the stack frame of current active function.
Return Value Register (AX): register for function return value.
(3) In every C program, main() is called by the C startup code crt0.o. When crt0.o calls main(), it pushes the return address (the current PC register) onto stack and replaces PC with the entry address of main(), causing the CPU to enter main(). For convenience, we shall show the stack contents from left to right. When control enters main(), the stack contains the saved return PC on top, as shown in Fig. 2.8, in which XXX denotes the stack contents before crt0.o calls main(), and SP points to the saved return PC from where crt0.o calls main().
(4) Upon entry, the compiled code of every C function does the following:
. push FP onto stack # this saves the CPU's FP register on stack.
. let FP point at the saved FP # establish stack frame
. shift SP downward to allocate space for automatic local variables on stack
. the compiled code may shift SP farther down to allocate some temporary working space on the stack, denoted by temps.

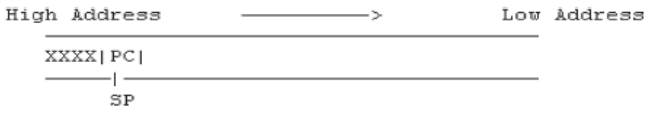
Process execution image

For this example, there are 3 automatic local variables, int x, y, z, each of sizeof (int) = 4 bytes. After entering main(), the stack contents becomes as shown in Fig, in which the spaces of x, y, z are allocated but their contents are yet undefined.

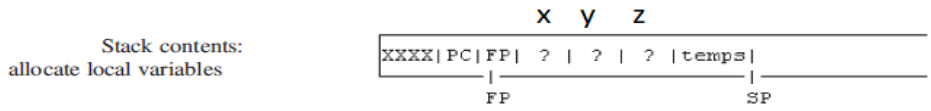
5. Then the CPU starts to execute the code x=10; y=20; z; which put the values 10, 20 into the memory locations of x, y, z, respectively. Assume that sizeof(int) is 4 bytes. The locations of x, y, c are at -4, -8, -12 bytes from where FP points at. These are expressed as -4(FP), -8(FP), -12(FP) in assembly code, where FP is the stack frame pointer. For example, in 32-bit Linux the assembly code for y=20 in C is

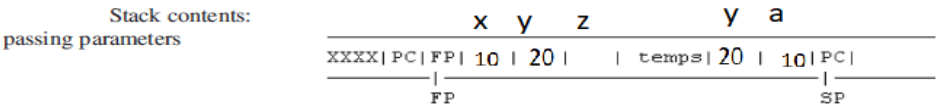
movl $20, -8(%ebp) # y=20 in C

main() calls add() by z=add(x, y); The compiled code of the function call consists of . Push parameters in reverse order, i.e. push values of y=20 and x=10 into stack. . Call add, which pushes the current PC onto stack and replaces PC with the entry address of add, causing the CPU to enter add().
When control first enters add(), the stack contains a return address at the top, preceded by the parameters, x, y, of the caller,

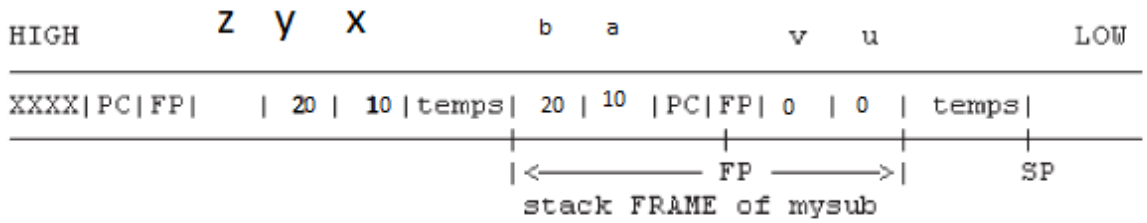


Stack contents in function call





Since add( ) is written in C, it actions are exactly the same as that of main(), i.e. it
. Push FP and let FP point at the saved FP; . Shift SP downward to allocate space for local variables v, u. The compiled code may shift SP farther down for some temp space on stack.
When add () executes the C statement return x+y+u+v, it evaluates the expression and puts the resulting value in the return value register (AX). Then it deallocates the local variables by



Upon return, the caller function catches the return value in the return register (AX). Then it cleans
The parameters a, b, from the stack (by adding 8 to SP). This restores the stack to the original
Situation before the function call. Then it continues to execute the next instruction

10. Many systems classify library functions as thread-safe or thread-unsafe. What causes a function to be unsafe for use by a multithreaded application? Implement Concurrent Hash Table that uses a lock per hash bucket each of which is represented by a list.

Adding locks to a data structure makes the structure **thread safe**.
A block of code is thread-safe if it can be simultaneously executed by multiple threads without causing problems.

- **Thread-safeness:** in a nutshell, refers an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions.
- For example, suppose that your application creates several threads, each of which makes a call to the same library routine:
    - This library routine accesses/modifies a global structure or location in memory.
    - As each thread calls this routine it is possible that they may try to modify this global structure/memory location at the same time.

- If the routine does not employ some sort of synchronization constructs to prevent data corruption, then it is not thread-safe.

Solution #1

- An obvious solution is to simply lock the list any time that a thread attempts to access it.
- A call to each of the three functions can be protected by a mutex.

Solution #2

- Instead of locking the entire list, we could try to lock individual nodes.
- A “finer-grained” approach.

```
// basic node structure
typedef struct __node_t {
int key;
struct __node_t *next;
pthread_mutex_t lock;
} node_t;
```

```
/* Implementation of a Concurrent Hash Table. threads-hash.c */
#include <stdio.h>
#include <stdlib.h>
#define HASH_BUCKETS (7)
typedef struct __node_t {
int key;
struct __node_t *next;
} node_t;
typedef struct __list_t {
node_t *head;
} list_t;
void List_Init(list_t *L) {
L->head = NULL;
}
void List_Insert(list_t *L, int key) {
node_t *new = malloc(sizeof(node_t));
if (new == NULL) { perror("malloc"); return; }
new->key = key;
new->next = L->head;
L->head = new;
}
int List_Lookup(list_t *L, int key) {
node_t *tmp = L->head;
while (tmp) {
if (tmp->key == key)
return 1;
tmp = tmp->next;
}
return 0;
}
void List_Print(list_t *L) {
node_t *tmp = L->head;
while (tmp) {
printf("%d ", tmp->key);
tmp = tmp->next;
}
printf("\n");
}
```

```
typedef struct __hash_t {
list_t hlists[HASH_BUCKETS];
pthread_mutex_t m[HASH_BUCKETS];
} hash_t;
void Hash_Init(hash_t *H) {
int i;
for (i = 0; i < HASH_BUCKETS; i++) {
List_Init(&H->hlists[i]);
}
}
void Hash_Insert(hash_t *H, int key) {
int b = key % HASH_BUCKETS;
List_Insert(&H->hlists[b], key);
}
void Hash_Print(hash_t *H) {
int i;
for (i = 0; i < HASH_BUCKETS; i++) {
printf("LIST %d: ", i);
List_Print(&H->hlists[i]);
}
}
int main(int argc, char *argv[]) {
hash_t myhash;
Hash_Init(&myhash);
Hash_Insert(&myhash, 10);
Hash_Insert(&myhash, 5);
Hash_Insert(&myhash, 30);
Hash_Print(&myhash);
return 0;
}
```

11. A 'big reader' lock provides multiple reader, single writer semantics optimized for workloads in which updates are rare. Implement 'big reader' software locks

```
Reader process:
Wait(mutex)
Rc++;
If(rc==1)
Wait(work);
Signal(mutex);
READ THE OBJECT
Wait(mutex);
Rc—
If(rc==0)
Signal(work);
Signal(mutex);
Writer process
Wait(work);
WRITE INTO THE OBJECT
Signal(work);
```

11.B. Is it necessary for a multiprocessor kernel to lock each variable or resource before accessing it? Enumerate the kinds of situations where a thread may access or modify an object without locking it. Write a program to demonstrate deadlock using mutex.

Yes. It is necessary for a multiprocessor kernel to lock each variable or resource before accessing it.

Locking not required when:

Thread not executing critical section. They are independent and not sharing any resources.

Program
Pthread mutext mutex;
Pthread mutext mutext 2, init;
Pthread mutex int (&mutex 2, NULL);
Void *dosomework-1(void * param)
{
Pthread mutex lock (&mutex1);
Pthread mutex lock(&mutex2);
DO SOME WORK
Pthread mutex unlock(&mutex2);
Pthread mutex unlock(&mutex 2);
Pthread exit(0);
}
Void * dosome_work_2(void *param)
{
Pthread mutex lock(&mutex 2);
Pthread mutex lock(& mutex &1);
DO SOME WORK
Pthread mutex unlock (&mutex 2);
Pthread mutex unlock(&mutex 1);
Pthread exit(0);

12a What issues must a programmer be concerned with in choosing an address to attach a shared memory region to? What errors would the operating system protect against? Why does the shared memory IPC structure acquire a reference to the anon_map for the segment? Illustrate shared memory data structures by giving algorithm for shmat()

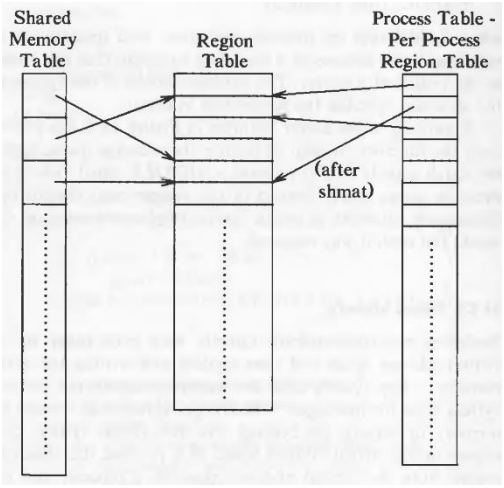
- **Check whether the hole between heap and stack segments is enough.**
- **where to attach the shared memory segment**
- **roundoff to page size**
- **protection of the shared memory segment**
- **fail if segment/region cannot be allotted**

Shared Memory

Sharing the part of virtual memory and reading to and writing from it, is another way for the processes to communicate.

```
shmid = shmget(key, size, flag);
```

where `size` is the number of bytes in the region. If the region is to be created, *allocreg* is used. It sets a flag in the shared memory table to indicate that the memory is not allocated to the region. The memory is allocated only when the region gets attached. A flag is set in the region table which indicates that the region should not be freed when the last process referencing it, *exit*s. The data structure are shown below:



Syntax for *shmat*:

```
virtaddr = shmat(id, addr, flags);
```

where addr is the virtual address where the user wants to attach the shared memory. And the flags specify whether the region is read-only and whether the kernel should round off the user-specified address. The return value *virtaddr* is the virtual address where the kernel attached the region, not necessarily the value requested by the process.
The algorithm is given below:

```
/*  Algorithm: shmat
 *  Input: shared memory descriptor
 *         virtual addresses to attach memory
 *         flags
 *  Output: virtual address where memory was attached
 */
{
        check validity of descriptor, permissions;
        if (user specified virtual address)
        {
                round off virtual address, as specified by flags;
                check legality of virtual address, size of region;
        }
        else                    // user wants kernel to find good address
                kernel picks virtual address: error if none available;
        attach region to process address space (algorithm: attachreg);
        if (region being attached for first time)
                allocate page tables, memory for region (algorithm: growreg);
        return (virtual address where attached);
}
```

If the address where the region is to be attached is given as 0, the kernel chooses a convenient virtual address. If the calling process is the first process to attach that region, it means that page tables and memory are not allocated for that region, therefore, the kernel allocated both using *growreg*.
The syntax for *shmdt*:

```
shmdt(addr);
```

where addr is the virtual address returned by a prior *shmat* call. The kernel searches for the process region attached at the indicated virtual address and detaches it using *detachreg*. Because the region tables have no back pointers to the shared memory table, the kernel searches the shared memory table for the entry that points to the region and adjusts the field for the time the region was last detached.

12. B. Consider the following snapshot of a system: Answer the following questions using the banker's algorithm: What is the content of the matrix Need? Is the system in a safe state? If a request from process P1 arrives for (0,4,2,0), can the request be granted immediately?

Need matrix:

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

With reference to Bankers algorithm
Need matrix is calculated by subtracting Allocation Matrix from the Max matrix

	Need(Max-Allocation)			
	A	B	C	D
P_0	0	0	0	0
P_1	0	7	5	0
P_2	1	0	0	2
P_3	0	0	2	0
P_4	0	6	4	2

To check if system is in a safe state

- The Available matrix is [1520]

- .
- A process after it has finished execution is supposed to free up all the resources it hold.
- We need to find a safety sequence such that it satisfies the criteria need $Need \leq Available$
- .
- Since $Need(P0) \leq Available$, we select $P0$.[$Available$]=[$Available$]+[$Allocation(P0)$]

$Available$=[1520]+[0012]=[1532]

- Need(P2) ≤ Available →

- Available=[1 5 3 2]+[1 3 5 4]=[2 8 8 6]
- Need(P3) ≤ Available→
- Available=[2 8 8 6]+[0 6 3 2]=[2 14 11 8]
- Need(P4) ≤ Available→
- Available=[2 14 11 8]+[0 0 1 4]=[2 14 12 12]
- Need(P1) ≤ Available→
  - Available=[2 14 12 12]+[1 0 0 0]=[3 14 12 12]
  - Safe Sequence is <p0,p2,p3,p4,p1>

A request from process P1 arrives for (0,4,2,0)

- System receives a request for P1 for $Req(P1)$[0420]

- • First we check if Req(P1) is less than $Need(P1) \rightarrow$[0420]<[0750]$istrue$
- • Now we check if Req(P1) is less than $Available \rightarrow$[0420]<[1520]$istrue$
- .
- So we update the values as:
  - $Available=Available-Request$=[1520]−[0420]=[1100]
- • $Allocation=allocation(P1)+Request$=[1000]+[0420]=[1420]
- • $Need=Need(P1)-Request$=[0750]−[0420]=[0330]

	Allocation				Max				Need				Available			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	0	0	0	0	1	1	0	0
P_1	1	4	2	0	1	7	5	0	0	3	3	0				
P_2	1	3	5	4	2	3	5	6	1	0	0	2				
P_3	0	6	3	2	0	6	5	2	0	0	2	0				
P_4	0	0	1	4	0	6	5	6	0	6	4	2				

- This is the modified table
- On verifying, we see that the safe sequence still remains the same .The system continues to remain in a safe state.