NAVIGATION

RESEARCH COMPUTING HOME
HPC HOME
NEWS
EVENTS
COMPUTATIONAL RESOURCES
HPC RESOURCE VIEW
SERVERS
STORAGE & QUOTA
OTHER HPC RESOURCES
FAQ: HPC RESOURCES
GETTING ACCESS TO HPC
ACCESS POLICY - GUEST/MEMBER
GETTING STARTED
GUI VISUAL ACCESS
HPC TUTORIALS
ON-DEMAND VIDEO TUTORIAL
COMMON STEPS BEFORE INSTALLING SOFTWARE
INSTALLING LOCAL R PACKAGES
INSTALLING LOCAL PYTHON MODULES

HPC Home > User Support/Getting Started >

Debugging Segmentation Faults

CONTENTS

Debugging

Code debugging is the process of detecting and eliminating compiler, runtime, and logical errors in the code. The debugger can be used in combination with valgrind like utility to detect and eliminate the source of segmentation faults. Each program (process) has access to certain portion of memory (user space or program space). During run time if the process

PDFmyURL

tries to access the memory outside its allocated space, the segmentation fault occurs. Segmentation faults are referred to as segfault, access violation or bus error. Hardware notifies the operating system about memory access violation. The OS kernel then sends the segmentation fault signal (SIGSEGV in Linux) to the process which caused the exception. The process then dump core and terminates. The core dump refers to the recorded state of the working memory of a computer program at a specific time when the program crashed. The state contains processor registers which may include the program counter and stack pointer, memory management information, and other processor and operating system flags and information. Core dumps are often used to assist in diagnosing and debugging errors in computer programs. Before proceeding to the Section B, the overview in Section A and Appendix A to D can be helpful.

For parallel Debugging, look for TotalView which has been installed in HPCC.

Segmentation Fault

Processes running in the system require a way to be informed about events that influence them. On UNIX there is infrastructure between the kernel and processes called **signals** which allows a process to receive notification about events important to it. When a signal is sent to a process, the kernel invokes a **handler** which the process must register with the kernel to deal with that signal. A **handler** is simply a designed function in the code that has been written to specifically deal with interrupt. When the kernel notices that you are touching memory outside your allocation, it will send you the **segmentation fault** signal.

Quick Guide

Compile with a 'g'(debug) flag:

Find the source file hello.c in Appendix A

```
gcc -g -o debug hello.c
```

(Note: debug mode is necessary to get the line number)

Debugging with Valgrind utlity:

```
valgrind --tool=memcheck --leak-check=yes -v --leak-check=full --show-reachable=yes ./debug
```

NAVIGATION

A. Process Flow for Execution of a Code

Case Study: Printing "Hello World" on the console – Linux Environment, x86_64 Architecture

```
C program: hello.c (Appendix A)
Compile: gcc hello.c
Executable: a.out
Generate assembly code: gcc -S hello.c
Assembly Code – hello.s (Appendix B):
```

Type the following in a bash shell:

```
./a.out
```

The linker creates an executable in a container format understood by the target Linux System – the Executable and Linking Format (ELF). The bash process makes a system call (Fig. 1) and the mode is changed from user to system (kernel). The fork system call creates a new task cloning the current process. The new process (child) gets unique PID and has the PID of the old parent process (PPID). The exec () system call replaces the entire current process with a new program i.e. bash replaced by a.out as showed in Fig. 1.

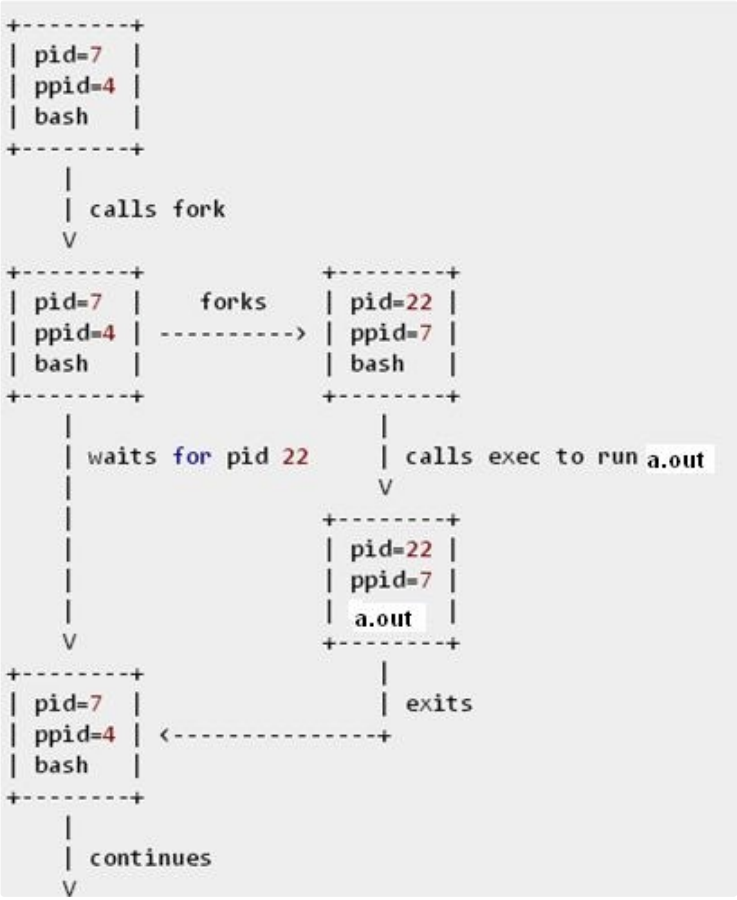


Fig. 1: Typical fork()/exec() operations where the bash shell is used to execute a.out [8]

At the end of the exec() system call, there is a new process entry of type task_struct in a process table, a circular doubly linked list showed in Fig. 2, waiting for scheduler to run; the wait time depending on the start timestamp in task_struct assigned by the kernel scheduling algorithm.

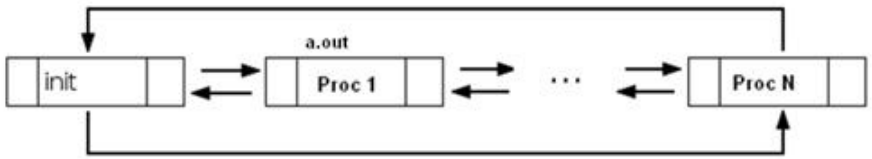


Fig. 2. Circular linked list representation of processes; init task's process descriptor (pointer to task_struct) is statically allocated.

Each process is assigned its own virtual memory as showed in Fig. 3. The major portion is allocated for user space and rest will be for kernel space. At any given period, process will not be using all of the code, data, and libraries. So, demand paging is used to map the portion of the virtual memory into physical memory when a process attempts to use. The process's page table is altered marking virtual areas as existing but not in memory. The pager only loads those pages that it expects the process to use right away into the physical memory; the other data in the address space are not available in the main memory. If the program attempts to access data not currently located in the main memory, there will be a page fault and operating system resolves it allowing the program to continue operation.

Fig.3: Process Virtual Memory space [9]

After the hardware timer interrupts when the current process (a.out) is assigned a CPU slice, the instruction cycle starts. The Program Control Block (PCB) data structure, consisting of the program counter and all of the CPU's registers as well as the process stacks containing temporary data, is loaded into CPU registers along with Program Counter (PC). In the fetch

state, the data pointed by the address in the program counter is fetched to the current instruction register (CIR). Initially, CPU fetch unit looks for data in the cache. If it is the first time, there will be no data in the cache and after the cache miss, the data is obtained from the system memory (DRAM). While loading the instruction in the instruction register, a circuit called memory cache controller loads a small block of data below the current position (address) that the CPU has just loaded (usually program flow is sequential) and the next position the CPU request will probably be the address immediately below the recent one. So, the next required data will probably reside in that block in the cache and more chances of cache hit.

Upon fetching the instruction, the program counter is incremented by one "address value" (to the location of the next instruction). Now, the instruction (op-code) in the CIR is decoded and the required data (operands) as interpreted are fetched to the data registers. The control unit passes decoded instruction to different parts of CPU (e.g. ALU) for execution. The results are stored in the main memory or sent to I/O devices.

If the peripheral components are involved to transfer their I/O data to and from main memory, DMA (Direct Memory Access) mechanism allows it without involving the CPU. The process is put to sleep during the transfer of data to DMA buffer and interrupt is used to awaken the process when it is time to read the data.

B. Debugging Segmentation Faults

The example code (test.cpp; Appendix E) has been compiled with GNU compiler and the segmentation fault has been detected using the valgrind utility. The valgrind can be used for intel compilers as well. Valgrind is a memory mismanagement detector which shows memory leaks, deallocation errors etc. The caveat is Valgrind can consume twice as much memory and takes longer to run the code. It can not detect the buffer overflow unless it causes stack overflow or corrupt the return address of the function.

Compile with a 'g'(debug) flag: g++ -g –o debug test.cpp

(Note: debug mode is necessary to get the line number)

Debugging with Valgrind utlity: valgrind --tool=memcheck --leak-check=yes -v --leak-check=full --show-reachable=yes ./debug

Stack Overflow

The variables, arrays, pointers, and arguments of a code reside in the General Purpose Registers and stack assigned for a process (program or user space). The stack has a fixed size. This is the normal mechanism whereby stack size adjusts to demand. However, if the maximum stack size has been reached, we have a stack overflow and the program receives a Segmentation Fault. For example, if the array is too big or the recursive function is consuming lots of stack space, the stack overflow occurs. The crossing of the boundary of the stack limit generates signal SIGSEGV causing segfault. As a solution, you may increase the size of the stack, however, dynamic allocation is the better way to resolve the issue.

Though Java or Java like languages throw “array out of bound” exception, C expects user to take care of it. It saves the space for the user that has been asked but does not check if the user is going out of bound. Array in the stack can overflow producing undefined values as long as there is not stack overflow or overwriting of the return address of the function leading to arbitrary code execution. There is a possibility of security threat as the return address can be replaced by the return address of another malicious program.

In debug.cpp, when the size of the array for b is huge (b[100000] = 10) at line 16, the following message is displayed:

```
Invalid write of size 4
==27229==    at 0x400696: main (test.cpp:16)
==27229==  Address 0x7ff0097f0 is not stack'd, malloc'd or (recently)
free'd
==27229==
==27229==
==27229== Process terminating with default action of signal 11 (SIGSEGV)
==27229==  Access not within mapped region at address 0x7FF0097F0
==27229==    at 0x400696: main (test.cpp:16)
==27229==  If you believe this happened as a result of a stack
==27229==  overflow in your program's main thread (unlikely but
==27229==  possible), you can try to increase the size of the
==27229==  main thread stack using the --main-stacksize= flag.
==27229==  The main thread stack size used in this run was 10485760.
```

If you use smaller size of array, let's say (b[100] = 10), there will be no segfault and you will get the output as 10. So, C/C++ does not check the array bound. Here, the left string prefixes ==27229== represents each line of Valgrind-output and the number 277229 is the PID of the processor.

Invalid Read

You are trying to access the value after it is freed. Here, in the first iteration, the node is freed "free(node)", however, it is tried to be accessed through "node->next" as showed:

```
struct vertex *next_node = NULL;
for(node = st[i]; node != NULL; ) {
next_node = node->next;
free(node);
sym = next_sym;
}
```

Also, you may have run out of the mapped region of memory address and hence the data can no longer be accessible. One scenario is showed below. The code snippet is changed from:

```
for (long int k = 0; k < NV ; k++) {
long int v_alternate = graph::getv()
v_alternate = graph::getv();
    if ( edge[vmin][vmin] > edge[v_alternate][v_alternate]){
...
   }
}
```

into:

```
long int v_alternate = graph::getv();
long int max_deg = edge[vmin][vmin];
long int min_deg = edge[v_alternate][v_alternate];

for (long int k = 0; k < NV ; k++) {
    if ( max_deg > min_deg){
...
   }
}
```

The valgrind output looks similar to:

```
==29196== Process terminating with default action of signal 11 (SIGSEGV)
==29196== Access not within mapped region at address 0xFFFFFFFFFFFFFFF8
==29196== at 0x401954: graph::graph(_IO_FILE*)
(vc_NOVCA_combined.cpp:243)
==29196== by 0x4025DE: main (vc_NOVCA_combined.cpp:504)
```

```
==29196== If you believe this happened as a result of a stack
==29196== overflow in your program's main thread (unlikely but
==29196== possible), you can try to increase the size of the
==29196== main thread stack using the --main-stacksize= flag.
==29196== The main thread stack size used in this run was 10485760.
```

Not Enough Memory Allocated in the heap

In the debug.cpp code, the memory is allocated for two integers (a = new int[2]) but its 100th integer location has been assigned a value (a[100]=4) for which no memory is allocated. The error is in line 15 of the code.

```
Invalid write of size 4

==17810==    at 0x400692: main (test.cpp:15)

==17810==  Address 0x4c2e1d0 is not stack'd, malloc'd or (recently)
free'd
```

Memory Leak

The memory leaks occur if the heap memory is not properly freed. In the test.cpp code, pointer ‘c’ is set to point to pointer ‘a’. So, “delete []c” cannot free the memory allocated (2* 4bytes) of memory, and hence the message below:

```
8 bytes in 1 blocks are definitely lost in loss record 1 of 1

==4699==    at 0x4A065BA: operator new[](unsigned long)
(vg_replace_malloc.c:264

)

==4699==    by 0x400687: main (test.cpp:11)

==4699==

==4699== LEAK SUMMARY:

==4699==    definitely lost: 8 bytes in 1 blocks
```

Again, the command delete []c is again trying to free the allocated memory which has already been freed using delete [] a, and hence the Invalid free() as showed below:

```
==4699== Invalid free() / delete / delete[]

==4699==    at 0x4A056AF: operator delete[](void*)
(vg_replace_malloc.c:368)

==4699==    by 0x40070C: main (test.cpp:25)

==4699==  Address 0x4c2e040 is 0 bytes inside a block of size 8 free'd

==4699==    at 0x4A056AF: operator delete[](void*)
(vg_replace_malloc.c:368)

==4699==    by 0x4006FC: main (test.cpp:24)
```

Uninitialized Values

If you try to print the value the variable contains which has not been previously assigned, the message looks like this:

```
==1022== Use of uninitialised value of size 8

==1022==    at 0x375C441DCD: _itoa_word (in /lib64/libc-2.5.so)

==1022==    by 0x375C4451B2: vfprintf (in /lib64/libc-2.5.so)

==1022==    by 0x375C44CFA9: printf (in /lib64/libc-2.5.so)

==1022==    by 0x4006C1: main (test.cpp:20)
```

In line 20 (printf("%d %d\n", a[0],b[5]), the value for b[5] which has never been assigned before, is attempted to be printed out. Again in line 23 (if (b[5]<2) b[4] = 5), the decision is based on the unassigned value, so you will be getting the following message:

```
==3157== Conditional jump or move depends on uninitialised value(s)

==3157==    at 0x4006DD: main (test.cpp:23)
```

Appendices

Appendix A

hello.c

```
/* Print Hello World */
#include <stdio.h>
int main ()
{
     printf ("Hello World"); //print Hello World in the console
   return 0; //Exit
}
```

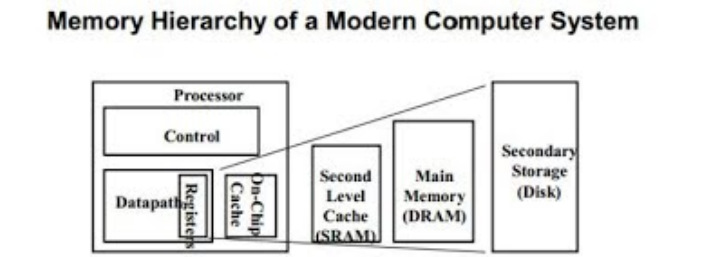
Appendix B

```
       .file   "test.c"
        .section        .rodata
.LC0:
        .string "Hello World"
        .text
.globl main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %eax
        movq    %rax, %rdi
        movl    $0, %eax
```

```
        call    printf
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (GNU) 4.4.7 20120313 (Red Hat 4.4.7-3)"
        .section        .note.GNU-stack,"",@progbits
```

Appendix C:

Memory Management



Level/Name	1/RF	2/C	3/MM	4/DM
Typical size	< 1 KB	< 16 MB	< 16 GB	> 100GB
Implementation technology	Custom memory w. multiple ports, CMOS	On-chip or off-chip CMOS SRAM	CMOS DRAM	Magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	5,000,000
Bandwidth	20,000-100,000 (MB/s)	5000-10,000 (MB/s)	1000-5000 (MB/s)	20-150 (MB/s)
Managed by	Compiler	Hardware	Operating system	OS/operator
Backed by	Cache	Main memory	Disk	CD or tape

Fig. Memory Hierarchy of a Modern Computer System [10]

CPU Registers:

Only few registers are available on the processor. For example, Intel Chips processor have 6 general purpose registers and specialized registers including a base register, stack register,

flag register, program counter, and addressing registers. Same speed as the rest of the CPU. They store the address of the currently executed instructions as well as data.

Register	Callee Save	Description
%rax		result register; also used in idiv and imul instructions.
%rbx	yes	miscellaneous register
%rcx		fourth argument register
%rdx		third argument register; also used in idiv and imul instructions.
%rsp		stack pointer
%rbp	yes	frame pointer
%rsi		second argument register
%rdi		first argument register
%r8		fifth argument register
%r9		sixth argument register
%r10		miscellaneous register
%r11		miscellaneous register
%r12-%r15	yes	miscellaneous registers

Fig. A1. The x86_64 General Purpose Register; 32-bit registers using the 'e' prefix.

```
long myfunc(long a, long b, long c, long d,
long e, long f, long g, long h)
{
long xx = a * b * c * d * e * f * g * h;
long yy = a + b + c + d + e + f + g + h;
long zz = utilfunc(xx, yy, xx % yy);
return zz + 20;
}
```

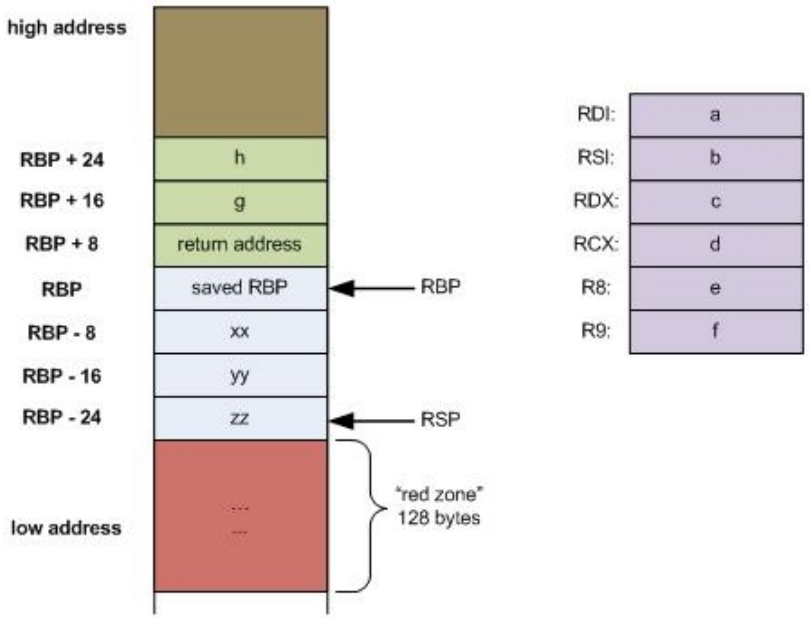


Fig. A2: Status of registers and stack frame for the C function myfunc; rbp pointing to the base of the frame whereas rsp pointing to the top of the frame; each register is 8bytes (64bits) long; stack growing from higher address to lower address [2].

Cache:

The cache works on two principles: temporal locality and spatial locality. The former is the idea that if you recently used a certain chunk of data, you'll probably need it again soon. The latter means that if you recently used the data at address X, you'll probably soon need address X+1. It means that you'll always want your own code to exploit these two forms of locality as much as possible without jumping all over memory. Working on one small area, and then move on to the next, improve the performance.

The cache is there to reduce the number of times the CPU would stall waiting for a memory request to be fulfilled (avoiding the memory latency), and as a second effect, possibly to reduce the overall amount of data that needs to be transferred (preserving memory bandwidth). Whenever data is to be read from main memory, the system hardware first checks for that data in the cache to speed up access. Different areas of RAMs are cached at different times through mapping as cache is much smaller space than main memory. When writing data from the CPU, the data is first written to cache before being written on main memory. Different levels of Cache – L1 located directly on the CPU chip, L2 is the part of CPU module, and L3 is the part of the system motherboard. The cache is made out of SRAM, a

solid state device. It is smaller but faster but requires more transistors per bit. SRAM does not need to be refreshed as the transistors (flip flops) inside would continue to hold the data as long as the power supply is not cut off. L1 cache is integrated in a CPU chip as is faster because of shorter data/address path.

Performance Improvement [14]:

· Use smaller data types

· Organize your data to avoid alignment holes (sorting your struct members by decreasing size is one way)

· Beware of the standard dynamic memory allocator, which may introduce holes and spread your data around in memory as it warms up.

· Make sure all adjacent data is actually used in the hot loops. Otherwise, consider breaking up data structures into hot and cold components, so that the hot loops use hot data.

· Avoid algorithms and data structures that exhibit irregular access patterns, and favor linear data structures.

Modern CPUs often have one or more hardware prefetchers. They train on the misses in a cache and try to spot regularities. For instance, after a few misses to subsequent cache lines, the hardware prefetcher will start fetching cache lines into the cache, anticipating the application's needs. If you have a regular access pattern, the hardware prefetcher is usually doing a very good job. And if your program doesn't display regular access patterns, you may improve things by adding prefetch instructions yourself.

Regrouping instructions in such a way that those that always miss in the cache occur close to each other, the CPU can sometimes overlap these fetches so that the application only sustain one latency hit (Memory level parallelism). To reduce the overall memory bus pressure, you have to start addressing what is called temporal locality. This means that you have to reuse data while it still hasn't been evicted from the cache. Merging loops that touch the same data (loop fusion), and employing rewriting techniques known as tiling or blocking all strive to avoid those extra memory fetches. While there are some rules of thumb for this rewrite exercise, you typically have to carefully consider loop carried data dependencies, to ensure that you don't affect the semantics of the program.

ROM:

ROM is a non-volatile memory where the BIOS resides i.e. ROM stores the initial program. BIOS contains all the codes required to control input/output devices (keyboard, disk drives,

communication) and miscellaneous functions. It makes it possible for the computer to boot; BIOS is copied from ROM to main memory known as shadowing as the main memory is faster than ROM.

Main Memory:

The data and programs that are being used are stored in main memory. It is a solid state device which is made of DRAM chips that have power connection, data connection, read/write signal connection, and address connection. Unlike SRAM, DRAM using capacitors requires data to be refreshed periodically in order to retain the data. The additional circuitry and timing for refreshing data makes DRAM memory slower than SRAM.

Virtual Memory:

There is never enough main memory so Virtual memory is the concept of combining main memory with the slower storage (hard drive) giving the system the appearance of having much more main memory than is actually available. The machine code for the application consumes some bytes on top of additional bytes for data storage and I/O buffers called application's address space. If the address space is more than the main memory, the application would not have run if there were no virtual memory.

The memory management hardware called Memory Management Unit (MMU) divides main memory into pages – contiguous sections of memory of a set size known as paging. The actual physical layout is controlled by process's page table. When a program is executed, process (with unique PID) is assigned for that program by the Kernel/OS. The executable image file consists of both executable code and data along with the information necessary to load the executable code and associated data into the virtual memory of the process. During the execution, processor can allocate memory to use which needs to be linked into processor's existing virtual memory. The shared libraries are also linked into the process's virtual address space along with other processes' virtual space. The great whole in the middle of the address space may never be used (e.g. arrays are often oversized and certain portions of the code are rarely used) unless the stack/heap grows to fill the gap.

At any given period, process will not be using all of the code, data, and libraries. So, demand paging is used to map the portion of the virtual memory into physical memory when a process attempts to use. The process's page table is altered marking virtual areas as existing but not in memory. The pager only loads those pages that it expects the process to need right away into the physical memory; the other data in the address space are not available in the main memory. If the program attempts to access data not currently located in the main memory, there will be a page fault and operating system resolves it allowing the program to continue operation. This new page will be included in a working set, a group of main memory

pages currently dedicated to the specific process. So, the working set grows with more page faults but shrinks when the pages are turned into free pages writing them into swapping space of mass storage device. The condition of excessive swapping is known as thrashing and indicates insufficient main memory for the present workload.

Each process has its own virtual memory (about 4gb) which maps to the physical memory through page tables. The virtual memory will split into major portion (about 3gb) for user space and small portion (about 1gb) for the use of kernel space. Kernel space is mapped to the starting locations of physical memory where the kernel image is loaded at the boot time. Besides the program instructions and data, the process also includes the program counter and all of the CPU's registers as well as the **process stacks** containing temporary data (routine parameters, return addresses, saved variables). The **heap** is an area of memory that is managed by the process for on the fly memory allocation. This is for variables whose memory requirements are not known at compile time.

The current context data is stored as process control block (PCB) data structure in Kernel space. When the process resumes (switching context), the program counter from PCB is loaded and execution continues. Each process, when created, is allocated with a new task_struct data structure in system (main) memory and is added into the task vector, an array of pointers pointing to every task_struct. The size of the task vector determines the maximum number of processes. Whilst the operating system can run many processes at the same time, in fact it only ever directly starts one process called the init process. This isn't a particularly special process except that its PID is always 0 and it will always be running.

Hard Drives:

Hard drive is non-volatile in nature i.e. data remains in it even after the power is removed. So, the programs and data for longer-use are stored in it. But the program has to be read into the main memory from hard drive to be executed. The hard drive is electro mechanical in nature and consists of phases – access arm movement, disk rotation, head reading/writing, and data transfer.

Off-line Backup Storage (tape, optical Discs):

This storage is usually for archiving data. The access time will be in seconds and capacity around Tera Byte to Peta Byte.

Operating System and Kernels

The kernel is the core of the operating system, a logic or program which has full access to all memory and hardware and acts as an interface between the user application and the hardware. User space program can not directly access system resources but Kernel handle the access on the program's behalf through system calls (e.g. printing data on the console; printf in C) allowing only well trusted code to run in the kernel mode/space. System call checks the arguments in the user program, builds the data structure to copy the arguments to the kernel, and executes the special instruction called software interrupt. The interrupt hardware of the CPU then saves the state of the user's program and switches the privilege level to the kernel mode. The kernel is responsible for memory management, process management, device management, interrupt handling, I/O communication, file systems, and scheduling process. The Kernel build along with the user friendly applications and utilities constitute the operating system.

The flow control during a boot is from BIOS, to boot loader, to kernel. The first stage is loaded and executed by the BIOS from the Master Boot Record (MBR) or another boot loader. The second stage of boot loader, when loaded and executed, displays GRUB startup menu allowing user to choose the operating system or examine/edit startup parameters. The boot loader loads the operating system (presented with boot options), sets up system functions (hardware, memory paging), and starts the kernel (start_kernel). The kernel then starts the program init (/sbin/init), the father of all processes, responsible mostly for running startup scripts for run level (/etc/rc.d/rc#.d with S (boot) or K(shutdown) prefix running in numerical order) presenting the user with a user space (login screen). Init creates processes from a script stored in the file /etc/inittab and then goes dormant waiting for three events to happen – processes end or die, power failure signal, or a request via /sbin/telinit to change the runlevel. The new processes may go on creating new processes (child process). For an example, the getty process creates a login process when a user attempts to login. All of the processes in the system are descended from the init kernel thread.

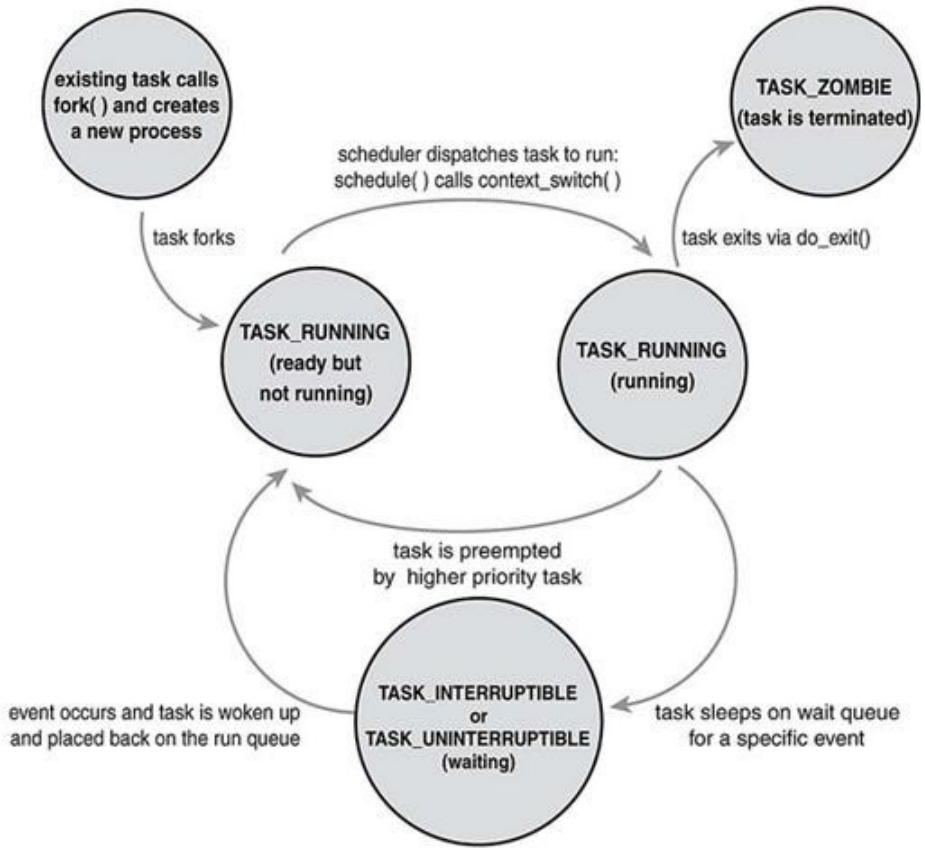


Fig. C1: Process States [7]

Direct Memory Access (DMA):

DMA allows certain peripheral devices or hardware subsystem within the computer to access the main memory without the need to involve the system processor eliminating computational (CPU) overhead. During time consuming I/O operations, CPU can perform other operations while the transfer is in the process with the process in sleep mode. DMA issues an interrupt when the operation is done to awaken the process. Hardware systems like disk drive controller, graphics cards, networks cards, and sound cards use DMA. DMA is also being used for intra-chip data transfer in multi-core processors and memory-to-memory copy.

Cached Memory in Linux:

Linux system makes use of various caches between the filesystems abstracted through Virtual Files System (VFS) and user level processes. VFS provides a uniform interface for the kernel for various I/O requests; the most important service is providing a uniform I/O data cache. The four caches of I/O data that Linux maintains are page cache, i-node cache, buffer cache, and directory cache as showed in Fig C2.

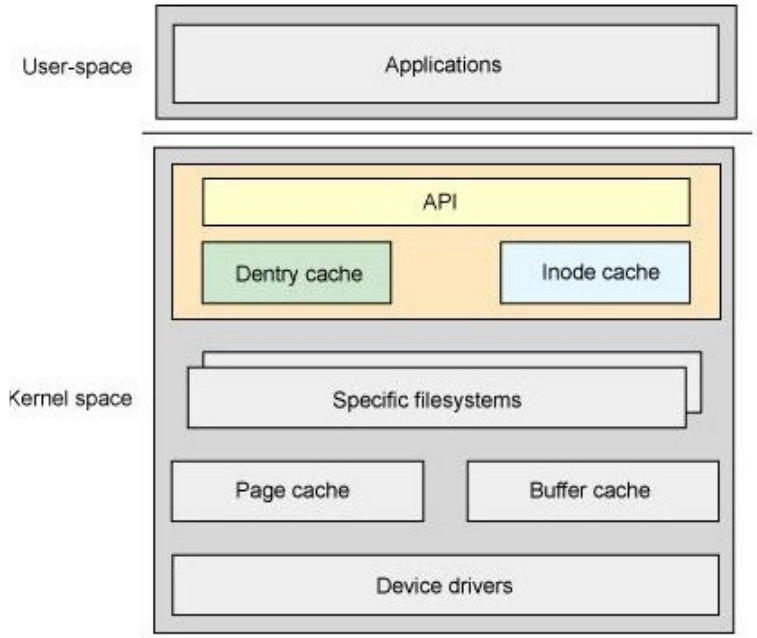


Fig. C2: Linux Kernel IO Cache [15]

Inode & Dentry:

Inode , an index node, is a data structure in Linux, which stores all the information (metadata information such as file ownership, access mode, file type etc) about a file system object (file, device node, socket, pipe, etc) but the data content and filename. Directory Entry (dentry): /usr/src/kernels/2.6.18-348.3.1.el5-x86_64/include/linux/dcache.h, manages the hierarchical nature of the file system with a root dentry and its child entries.

Page Cache:

Page cache accelerates the file access by storing the data in unused areas of system memory called cached memory during first file read from hard drives or write to them. When the data

is read again later, it can be read quickly from the cache. Page cache combines virtual memory and file data.

Example:

Execution time for link list (ll) command (time ll) for the first time is shorter than the subsequent ones as showed below:

```
real 0m0.011s
user 0m0.001s
sys 0m0.004s
```

```
real 0m0.006s
user 0m0.001s
sys 0m0.004s
```

Also, the amount of cache increases after the first ll as showed below (Cached Memory in Ganglia):

```
free -m
```

output:

```
total used free shared buffers cached
Mem: 48291 46689 1602 0 40 41103
```

```
free -m
```

output:

```
 total used free shared buffers cached
Mem: 48291 46689 1602 0 40 41336
```

If the data is written to a file, it is first written to the page cache stored as dirty pages which is periodically transferred or transferred through system call such as sync or fsync.

Inode Cache:

Iode cache speeds up the access of the mounted file system. When the mounted file systems are navigated, the VFS inodes are continually read and written (in some cases).

Dentry Cache:

When the directories are looked up by the real file system, their details are added to the directory cache to speed up the access the next time. Since only the short directory entries are cached (up to 15 characters, short directory names are helpful).

Buffer Cache:

Device file or the block device is an interface for the device driver (computer program that operates or controls a particular device attached to the computer such as flash drive) that appears in a file system (/dev) as if it were an ordinary file. In case of a flash drive, if there is a need to access the data, all block data read and write requests are given to the flash drivers in the form of data structures via standard kernel system calls. The device identifier uniquely identifies the device and the block number tells the driver which block to read. To speed up access, Linux maintains the buffer cache to save the information.

Appendix D

Object File

Objdump command in Linux is used to provide thorough information on object files. It can be a very handy tool for normal programmers during debugging.

Command: $ objdump -x -d -S a.out

Output:

```
a.out:     file format elf64-x86-64  # The file is in ELF format

a.out

architecture: i386:x86-64, flags 0x00000011:

HAS_RELOC, HAS_SYMS

start address 0x0000000000000000


Sections:

Idx Name          Size      VMA               LMA               File
off  Algn

  0 .text         0000001d  0000000000000000  0000000000000000
00000040  2**2

                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE

  1 .data         00000000  0000000000000000  0000000000000000
00000060  2**2

                  CONTENTS, ALLOC, LOAD, DATA
```

```
  2 .bss          00000000  0000000000000000  0000000000000000
00000060  2**2
                  ALLOC
  3 .rodata       0000000c  0000000000000000  0000000000000000
00000060  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      0000002d  0000000000000000  0000000000000000
0000006c  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  0000000000000000  0000000000000000
00000099  2**0
                  CONTENTS, READONLY
  6 .eh_frame     00000038  0000000000000000  0000000000000000
000000a0  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 test.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .rodata        0000000000000000 .rodata
0000000000000000 l    d  .note.GNU-stack        0000000000000000
.note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     F .text  000000000000001d main
0000000000000000         *UND*  0000000000000000 printf
Disassembly of section .text:


0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   b8 00 00 00 00          mov    $0x0,%eax
```

```
                        5: R_X86_64_32  .rodata
   9:   48 89 c7                mov    %rax,%rdi
   c:   b8 00 00 00 00          mov    $0x0,%eax
  11:   e8 00 00 00 00          callq  16 <main+0x16>
                        12: R_X86_64_PC32       printf-0x4
  16:   b8 00 00 00 00          mov    $0x0,%eax
  1b:   c9                      leaveq
  1c:   c3                      retq
```

Here,

Size: size of the loaded section

VMA: Virtual Memory Address

LMA: Logical Memory Address

off: offset from the beginning of the file

The symbols in the object file as a part of inspection can be done using readelf (Read Executable and Linkable format)

```
readelf --symbols ./hello.o
```

output:

```
Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND
printf@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND
__libc_start_main@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND
__gmon_start__


Symbol table '.symtab' contains 63 entries:
```

```
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000400238     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000400254     0 SECTION LOCAL  DEFAULT    2
...
```

Procedure Lookup Table

```
readelf --relocs ./hello
```

output:

```
Relocation section '.rela.dyn' at offset 0x368 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name
+ Addend
000000600ff8  000300000006 R_X86_64_GLOB_DAT 0000000000000000
__gmon_start__ + 0


Relocation section '.rela.plt' at offset 0x380 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name
+ Addend
000000601018  000100000007 R_X86_64_JUMP_SLO 0000000000000000
printf@GLIBC_2.2.5 + 0
000000601020  000200000007 R_X86_64_JUMP_SLO 0000000000000000
__libc_start_main@GLIBC_2.2.5 + 0
```

Appendix E:

test.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main ()
```

```
{
    //char *x = new char[100];
    int *a;
    int *c;
    int b[10];
    a  = new int[2];
    //c = new int[2];
    // int a[2]; // even this is not giving error
//    c = a;
    a[0] = 1000;
   // a[100] = 4;
    b[100] = 10;
    b[4] = 15;
    b[3] = 25;
printf ("Hello World\n");
printf("%d %d\n", a[0],b[3]);
printf("%d\n",b[100]);

if (b[5]<2) b[4] = 5;
   delete [] a;
//   delete[] c;
   return 0;
}
```

References:

[1] Physical and Virtual Memory: http://www.centos.org/docs/4/html/rhel-isa-en-4/ch-memory.html

[2] Stack Frame Layout: http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/

[3] X86 Assembly: http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax

[4] Linux Objedump Command: http://www.thegeekstuff.com/2012/09/objdump-examples/

[5] Understanding the Linux Kernel: http://oreilly.com/catalog/linuxkernel/chapter/ch10.html

[6] Processes: http://tldp.org/LDP/tlk/kernel/processes.html

[7] Process States: http://www.informit.com/articles/article.aspx?p=370047

[8] Exec() and fork():http://stackoverflow.com/questions/1653340/exec-and-fork

[9] Process Virtual Memory Space: http://www.princeton.edu/~unix/Solaris/troubleshoot/vm.html

[10] Memory Hierarchy: http://www.ece.eng.wayne.edu/~czxu/ece7660_f05/cache-basics.pdf

[11] Direct Memory Access: http://lwn.net/images/pdf/LDD3/ch15.pdf

[12] Valgrind: http://www.cprogramming.com/debugging/valgrind.html

[13] Debugging: http://www.cprogramming.com/debugging/segfaults.html

[14] Cache: http://stackoverflow.com/questions/763262/how-does-one-write-code-that-best-utilizes-the-cpu-cache-to-improve-performance

[15] Linux Cached Memory: http://www.ibm.com/developerworks/library/l-virtual-filesystem-switch/#resources

Comments

You do not have permission to add comments.

Sign in | Recent Site Activity | Report Abuse | Print Page | Powered By **Google Sites**

PDFmyURL.com - convert URLs, web pages or even full websites to PDF online. Easy API for developers!