Module - Virtual Address Space

Programs

Source program: Program written in high-level language or Assembly language. It may be a single file program or a multi-file program. Typically we use an editor (vi, emacs) to write source program.

Object program: Machine language code generated after compiling source program.

Executable program: Program generated after linking object program file(s) with dependent libraries (eg. libc, libm, librt, …). It contains machine language code. Executable programs are machine dependent. Executable programs are also called binary program.

Source program:
Open a terminal and write the below program in sample.c using your preferred editor.
```
$ vi sample.c
#include <stdio.h>
int main()
{
  char ch;
  printf("hello world\n");
  scanf("%c", &ch);
}
```
Object program:
To generate object program from source program, use the compiler "gcc" [*] with -c option.
```
$ gcc -o sample.o -c sample.c
$ file sample.o
sample.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV),
          not stripped
```
Executable program:
To generate executable program from object programs & libraries, use linker "ld" [†]
```
$ ld  -o sample -lc sample.o
$ file sample
sample: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
        dynamically linked (uses shared libs), for GNU/Linux
        2.6.18, not stripped
```
Here -lc refers to libc. libc [‡] contains executable code for all C standard library routines.
To generate executable program from source programs
```
  $ gcc -o sample sample.c
```

[*] gcc can be found in /usr/bin/.
[†] ld can be found in /usr/bin/
[‡] libc can be found in /lib/ or /lib64/

In this case compiler internally calls linker.

Programs (contd…)

Statically linked executable: An executable program which includes copy of binary code of referenced functions from all dependent libraries. Statically linked executables are bigger in file size due to the copy of binary code from libraries. They make applications non portable across solaris releases. Note: LINUX doesn't recommend statically linked executables.

Dynamically linked executable: An executable program which includes just a reference to binary code of referenced functions from all dependent libraries. Dynamically linked executables are relatively smaller in file size. They make applications portable across Linux releases. This is the default linkage option in Linux.

Statically linked executable Program:
To generate statically linked executable program use gcc's "-static" option. Linux doesn't recommend statically linked executables.

```
  $ gcc -static -o sample_s sample.c
  $ file sample_s
  sample_s: ELF 64-bit LSB executable, AMD x86-64, version 1 (SYSV),
            for GNU/Linux 2.4.0, statically linked, not stripped
```

To list all libraries required by a binary, use ldd [§]command.

```
  $ ldd sample_s
  not a dynamic executable
```

Dynamically linked executable Program:
To generate dynamically linked executable program don't use gcc's "-static" option.

```
  $ gcc -o sample_d sample.c
  $ file sample_d
  sample_d: ELF 64-bit LSB executable, AMD x86-64, version 1 (SYSV),
            for GNU/Linux 2.4.0, dynamically linked (uses shared
            libs), not stripped
```

To list all libraries required by a binary, use ldd command.

```
   $ ldd sample_d
       libc.so.6 => /lib64/tls/libc.so.6 (0x0000003bce900000)
       /lib64/ld-linux-x86-64.so.2 (0x0000003bce500000)
```

[§] ldd can be found in /usr/bin/

Programs (contd…)

Debug Executable Program: An executable program with debug symbols included in it. Debuggers can only work well with Debug executable programs. Debug information are

- Symbol Table
- Line number and source filename corresponding to each symbols to facilitate source level debugging
- Other debug information for debugger

Debug executables are bigger in file size due to extra debug information. These kind of executables are used before production.

Stripped Executable Program: An executable program from which debug symbols got removed. These kind of executables are used in production environment. Stripping removes

- Symbol Table
- Line number and source filename corresponding to each symbols
- Other debug information

Note: “compiling without -g option of gcc” and “compiling with -g option of gcc followed by strip” are not equivalent. gcc without -g produces a non stripped binary.

Debug Executable Program:
To generate executable program with debugging information, use gcc’s “-g” option.

```
 $ gcc -g -o sample_debug sample.c
 $ file sample_debug
 sample_debug: ELF 64-bit LSB executable, AMD x86-64, version 1
              (SYSV), for GNU/Linux 2.4.0, dynamically linked (uses
              shared libs), not stripped
```

Stripped Executable Program:
To strip debugging information from an executable program, use strip [**] command.

```
 $ cp sample_debug sample_stripped
 $ strip sample_stripped
 $ file sample_stripped
 sample_stripped: ELF 64-bit LSB executable, AMD x86-64, version 1
                  (SYSV), for GNU/Linux 2.4.0, dynamically linked
                  (uses shared libs), stripped
```

[**] Strip command can be found in /usr/bin/

Multi-file programs

Multi-file program: A Source program may be a single file program or a multi-file program.

- Single file programs restricts code reuse.
- Multi-file programs allow code reuse at function, macro.
- Different files could be developed by different people.
- Set of related files are compiled into Libraries.

Write the below code in `fact.h` file using your preferred editor.

```
$ vi fact.h
int fact(int n);
```

Write the below code in `fact.c` file using your preferred editor.

```
$ vi fact.c
#include "fact.h"

int fact(int n)
{
  int i, f;

  for(f=1, i=1; i<=n; i++) {
    f = f * i;
  }
  return f;
}
```

Write the below code in `mainfact.c` using your preferred editor.

```
$ vi mainfact.c
#include <stdio.h>
#include "fact.h"

int main()
{
  int n, f;
  char ch;

  printf("To calculate factorial, enter n:", &n);
  scanf("%d", &n);
```

```
    f = fact(n);
    printf("n!=%d\n", f);
    scanf("%c", &ch);
    return 0;
  }
```

Compile the above file.

```
  $ gcc -o mainfact.o -c mainfact.c
```

Link both the object programs `fact.o` , `mainfact.o`, and `libc` to create executable program.

```
  $ gcc -o mainfact -lc mainfact.o fact.o
```

Run the program

```
  $ ./mainfact
  To calculate factorial, enter n:5
  n!=120
```

Libraries

Libraries: A set of predefined general purpose functions packaged in a file is called a library. Eg. printf(), scanf(), fopen(), fclose() etc, are standard C functions defined in a library called libc. sin(), cos(), pow() etc are mathematical functions defined in libm. pthread_create(), pthread_destroy(), etc are defined in libpthread.

There are two kinds of binaries as far as linking to libraries are concerned. Correspondingly there are two kinds of libraries

Statically linked binaries: An executable program in which all library functions are part of. Statically linked binaries are bigger in size. Libraries that are used to do static linkage are called static libraries.

Dynamically linked binaries: An executable program in which all library functions are linked just before the execution. Dynamically linked binaries are smaller in size. Libraries that are used to do dynamic linkage are called dynamic libraries.

```
$ ls -l /usr/lib/libc*.a
-rw-r--r--. 1 root root   20388 2010-04-16 18:52 /usr/lib/libc_nonshared.a
```

Note: Recent linux distributions dont support / recommend statically linked binaries

```
$ ls -l /usr/lib/libc.*
-rw-r--r--. 1 root root 238 2010-04-16 18:21 /usr/lib/libc.so
```

Processes

Process:

- A running instance of executable program.
- process exists only in RAM, where as executable program exists on disk.
- Each running instance of same executable program is a separate & independent process.
- Each process has a unique process id (a +ve number)
- Each process has a parent process (except the init process, which is grandest parent in the system)
- Each process has a separate virtual address space.

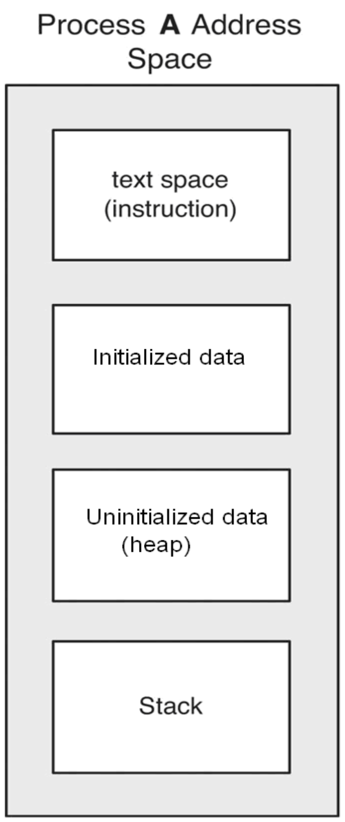
Address space:

Each process's virtual address space has three types of segments:

- Text segment (Program code & Shared library code)
- Initialized data segment
- Uninitialized data segment
- Stack segment

.

Process (contd…)



Text segment: Text segment contains program's binary code and libraries' binary code. Statically linked executables have bigger text segment. And dynamically linked executables have smaller text segment.

Data segment: Data segment contains two parts. The Initialized data , which contains global variables. And Uninitialized data which contains dynamically allocated variables (also called heap). Initialized data segment cannot grow and shrink, while a process runs. Where as uninitialized data segment can grow and shrink while a process runs, due to dynamic memory allocations and de-allocations. Initialized data segment is sometimes called Anonymous data segment.

Stack segment: Stack contains Activation records (also called stack frames) of functions. Stack frame typically contains return address, contents of registers, processor status word, return value pointer, arguments, and local variables in a function. When a function gets executed its stack frame is pushed on the top of the Stack segment. In other words, If a function called from another function, a stack frame for the called function is pushed onto stack segment. Essentially, a stack segment can grow & shrink due to call and return from a function respectively. For eg, Recursion repeatedly pushes same stack frame onto the stack segment.

Process (contd…)

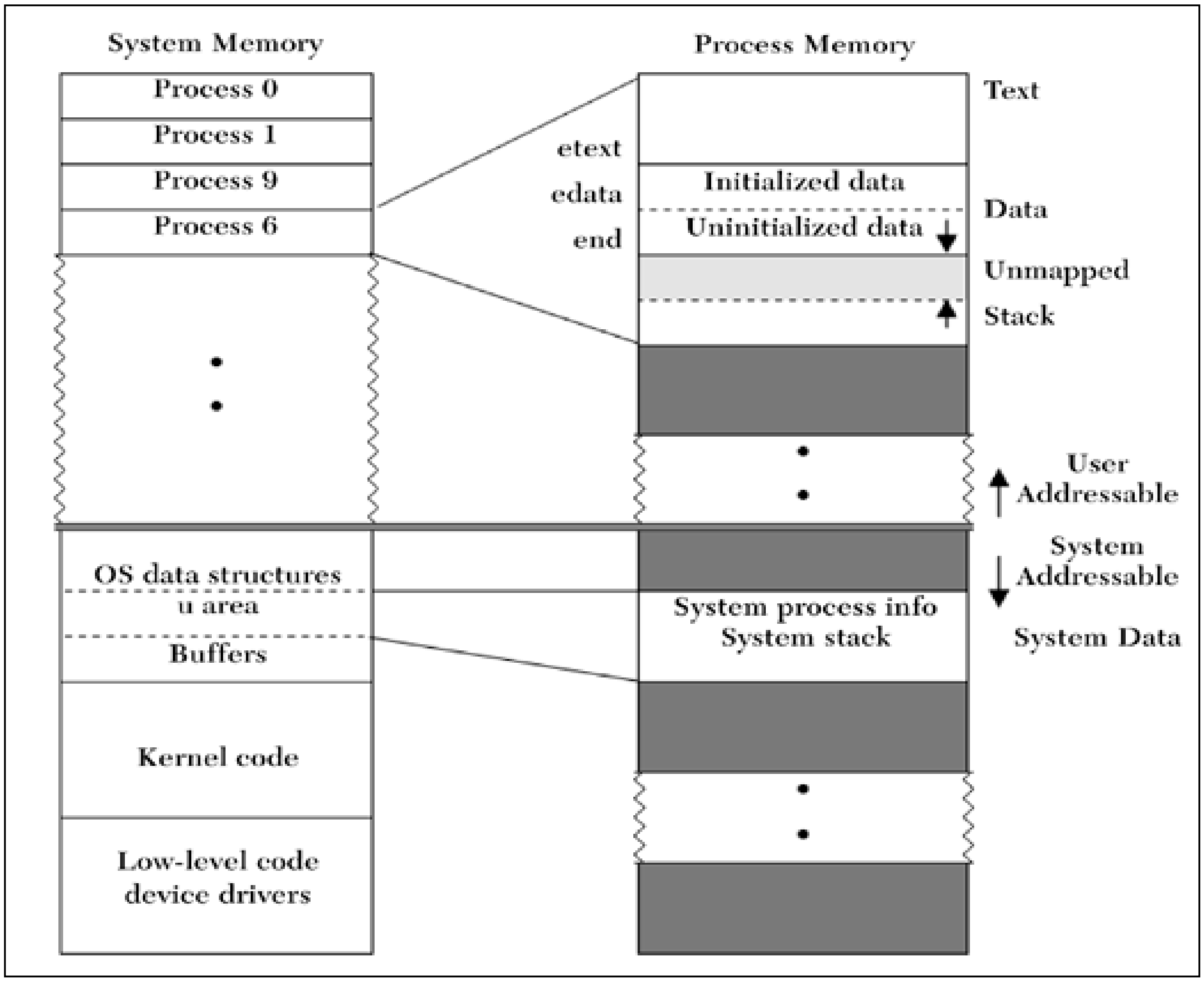


Figure above shows system memory, containing processes in user space and OS data structures, u-area, buffers, kernel code, device drivers code in kernel space. Figure also shows zoom-in of a process's virtual address space.

Text segment: Text and etext represent begin and end of Text segment respectively. Text segment cannot grow after programs starts running. Exception to this is dynamic loading libraries (DLLs), where text segment can also grow and shrink due to load and unload of libraries.

Data segment: Data and edata represent begin and end of Data segment respectively.

Stack segment: Stack represents top (begin) of stack segment. Note that stack grows from bottom to top. So stack segment's gets changed due to growth and shirking.

Virtual address space of process

Open a terminal and compile the sample program from the previous section.

```
$ gcc –o sample sample.c
```

In the same terminal run the sample program. The program waits for a character input due to the scanf(). Don't enter any input, so that process doesn't exit.

```
$ ./sample
hello world
```

Open second terminal and find process id of the sample process.

```
$ ps -af -U maruthisi
maruthisi  6131  2569  0 05:41 pts/0    00:00:00 ./sample
```

sample's process id is 6131. Note you might see many other processes in the output.

In the second terminal, run cat /proc/<pid>/status and cat /proc/<pid>/maps commands. Where <pid> is the above process id.

```
$ cat /proc/6131/status
Name:   sample
State:  S (sleeping)
Tgid:   6131
Pid:    6131
PPid:   2569
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
Utrace: 0
FDSize: 256
Groups: 1000
VmPeak:     3892 kB
VmSize:     3764 kB
VmLck:         0 kB
VmHWM:       408 kB
VmRSS:       408 kB
VmData:       48 kB
VmStk:        84 kB
VmExe:         4 kB
VmLib:      1548 kB
VmPTE:        28 kB
Threads:        1
SigQ:   2/16117
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: ffffffffffffffff
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed:
00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,0000
0000,00000000,00000000,00000000,00000000,00000000,00000000,00000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        2
nonvoluntary_ctxt_switches:     1
```

In the above output, key fields of our interest are:

* VmPeak: Peak virtual memory size.
* VmSize: Virtual memory size.
* VmData, VmStk, VmExe: Size of data, stack, and text segments.
* VmLib: Shared library code size.
* Threads: Number of threads in process containing this thread.

```
$ cat /proc/6131/maps

00400000-00401000         r-xp 00000000 08:03 295231 /home/maruthisi/sample
00600000-00601000         rw-p 00000000 08:03 295231 /home/maruthisi/sample
7f0ce7bde000-7f0ce7d42000 r-xp 00000000 08:03 234798  /lib64/libc-2.10.1.so
7f0ce7d42000-7f0ce7f42000 ---p 00164000 08:03 234798  /lib64/libc-2.10.1.so
7f0ce7f42000-7f0ce7f46000 r--p 00164000 08:03 234798  /lib64/libc-2.10.1.so
7f0ce7f46000-7f0ce7f47000 rw-p 00168000 08:03 234798  /lib64/libc-2.10.1.so
7f0ce7f47000-7f0ce7f4c000 rw-p 00000000 00:00 0
7f0ce7f4c000-7f0ce7f6b000 r-xp 00000000 08:03 234791  /lib64/ld-2.10.1.so
7f0ce8144000-7f0ce8146000 rw-p 00000000 00:00 0
7f0ce8166000-7f0ce816a000 rw-p 00000000 00:00 0
7f0ce816a000-7f0ce816b000 r--p 0001e000 08:03 234791  /lib64/ld-2.10.1.so
7f0ce816b000-7f0ce816c000 rw-p 0001f000 08:03 234791  /lib64/ld-2.10.1.so
7fff442e7000-7fff442fc000 rw-p 00000000 00:00 0       [stack]
7fff443f6000-7fff443f7000 r-xp 00000000 00:00 0       [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Example: heap of a process

Open a terminal write the below program using your preferred editor.

```
$ vi heap.c

#include <stdio.h>
#include <stdlib.h>

int main()
{
  char ch;
  void *ptr;
  scanf("%c",&ch);
  ptr = malloc(1048576); // 1MB
  scanf("%c",&ch);
  free(ptr);
}
```

Compile the heap program

```
$ gcc -o heap heap.c
```

Run the program in the same terminal. The program waits for a character input due to the scanf(). Don't enter any input.

```
$ ./heap
```

Open second terminal and find process id of the heap process.

```
$ ps -af -U maruthisi
UID         PID  PPID   C STIME TTY          TIME CMD
maruthisi  2912  2739  0 06:41 pts/1    00:00:00 ./heap
```

In the second terminal, run "`cat /proc/<pid>/status`" command with the above process id. Note that VmData shows the size of Heap segment.

```
$ cat /proc/2912/status
Name:   heap
State:  S (sleeping)
Tgid:   2912
Pid:    2912
PPid:   2739
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
Utrace: 0
FDSize: 256
Groups: 1000
VmPeak:     3892 kB
VmSize:     3760 kB
VmLck:         0 kB
VmHWM:       368 kB
VmRSS:       368 kB
VmData:       44 kB
VmStk:        84 kB
VmExe:         4 kB
VmLib:      1548 kB
VmPTE:        32 kB
Threads:        1
SigQ:   2/16117
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
```

```
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: ffffffffffffffff
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed:
00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,0000
0000,00000000,00000000,00000000,00000000,00000000,00000000,00000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        2
nonvoluntary_ctxt_switches:     0
```

In the first terminal, enter a key for the first scanf. The program does malloc() and waits for input for next scanf(). Don't enter any input.

In the second terminal, run "`cat /proc/<pid>/status`". Notice the increase in size of heap segment.

```
$ cat /proc/2912/status
Name:   heap
State:  S (sleeping)
Tgid:   2912
Pid:    2912
PPid:   2739
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
Utrace: 0
FDSize: 256
Groups: 1000
VmPeak:     4788 kB
VmSize:     4788 kB
VmLck:         0 kB
VmHWM:       416 kB
VmRSS:       416 kB
VmData:     1072 kB
VmStk:        84 kB
VmExe:         4 kB
VmLib:      1548 kB
VmPTE:        32 kB
Threads:        1
SigQ:   2/16117
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: ffffffffffffffff
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed:
00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,0000
0000,00000000,00000000,00000000,00000000,00000000,00000000,00000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        3
nonvoluntary_ctxt_switches:     0
```

Example: stack of a process.

```
$ vi stack.c

#include <stdio.h>

char ch;
void func()
{
  char var[1048576]; // 1MB
  scanf("%c",&ch);
}

int main()
{
  printf("%c",&ch);
  scanf("%c",&ch);
  func();
}
```

Compile the stack program

```
$ gcc -o stack stack.c
```

Run the program from one terminal. When the program waits at the first scanf(), don't enter any key.

```
$ ./stack
```

Find the process id of the process from second terminal.

```
$ ps -af -U maruthisi
UID        PID  PPID  C STIME TTY          TIME CMD
maruthisi  2936  2739  0 06:50 pts/1    00:00:00 ./stack
```

Process id is 2936. See address space of the process using "cat /proc/<pid>/status" from second terminal. Note that VmStk shows the size of stack segment.

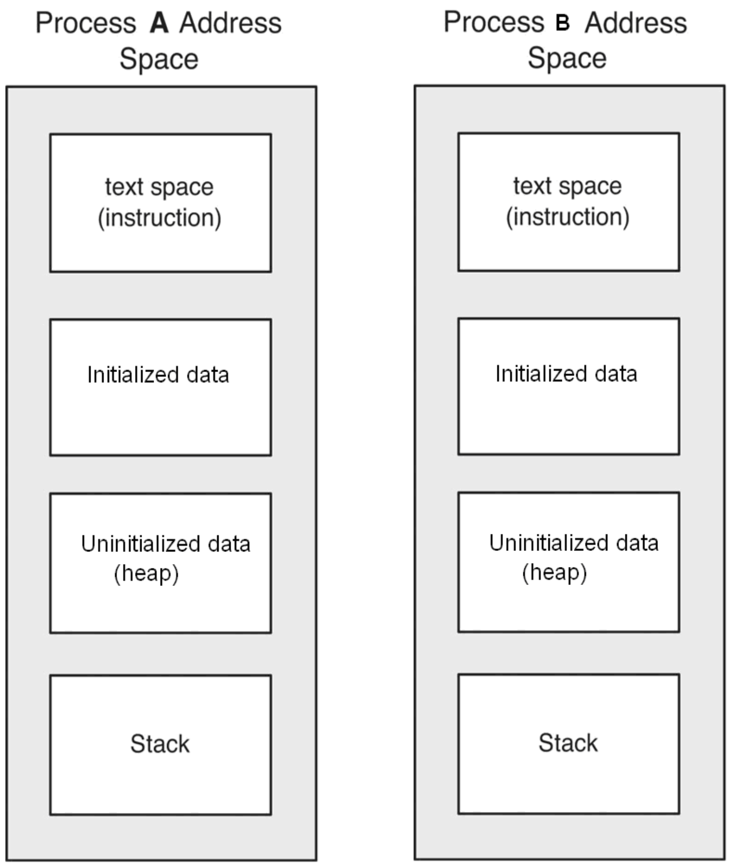
```
$ cat /proc/2936/status
Name:   stack
State:  S (sleeping)
Tgid:   2936
Pid:    2936
PPid:   2739
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
Utrace: 0
FDSize: 256
Groups: 1000
VmPeak:     3892 kB
VmSize:     3760 kB
VmLck:         0 kB
VmHWM:       368 kB
VmRSS:       368 kB
VmData:       44 kB
VmStk:        84 kB
VmExe:         4 kB
VmLib:      1548 kB
VmPTE:        32 kB
Threads:        1
SigQ:   2/16117
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
```

```
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: ffffffffffffffff
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed:
00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,0000
0000,00000000,00000000,00000000,00000000,00000000,00000000,00000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        1
nonvoluntary_ctxt_switches:     0
```

Now, press any key in the first terminal. After the function was called, address space shows growth in stack.

```
$ cat /proc/2936/status
Name:   stack
State:  S (sleeping)
Tgid:   2936
Pid:    2936
PPid:   2739
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
Utrace: 0
FDSize: 256
Groups: 1000
VmPeak:     4708 kB
VmSize:     4708 kB
VmLck:         0 kB
VmHWM:       380 kB
VmRSS:       380 kB
VmData:       44 kB
VmStk:      1032 kB
VmExe:         4 kB
VmLib:      1548 kB
VmPTE:        32 kB
Threads:        1
SigQ:   2/16117
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: ffffffffffffffff
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed:
00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,0000
0000,00000000,00000000,00000000,00000000,00000000,00000000,00000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        2
nonvoluntary_ctxt_switches:     0
```

Multiple processes of same program.



When multiple processes of same program are there, operating system creates separate text, data, heap, stack segments in their respective virtual address spaces. Even though at the virtual memory subsystem level, text segments for different process of same program can be mapped to same physical page. But data, heap, stack segments have to maintained separate in separate physical pages.

In other words, text segments can be mapped[††] in shared mode read-only, but data, heap, stack have to mapped in private writable mode.

[††] in unix mapping & unmapping is done using mmap() & munmap() system calls respectively.

Virtual address spaces of multiple processes of same program.

Run two instances the sample from previous section in two different shells and look for the PIDs from third shell.

```
$ ps -af -U maruthisi
UID        PID  PPID  C STIME TTY          TIME CMD
maruthisi  3027 2739  0 07:05 pts/1    00:00:00 ./sample
maruthisi  3028 2994  0 07:05 pts/3    00:00:00 ./sample
```

See virtual address space of the processes using “cat /proc/<pid>/status”.

```
$ cat /proc/3027/status
Name:   sample
State:  S (sleeping)
Tgid:   3027
Pid:    3027
PPid:   2739
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
Utrace: 0
FDSize: 256
Groups: 1000
VmPeak:     3892 kB
VmSize:     3764 kB
VmLck:         0 kB
VmHWM:       380 kB
VmRSS:       380 kB
VmData:       48 kB
VmStk:        84 kB
VmExe:         4 kB
VmLib:      1548 kB
VmPTE:        32 kB
Threads:        1
SigQ:   2/16117
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: ffffffffffffffff
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed:
00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,0000
0000,00000000,00000000,00000000,00000000,00000000,00000000,00000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        2
nonvoluntary_ctxt_switches:     0

$ cat /proc/3028/status
Name:   sample
State:  S (sleeping)
Tgid:   3028
Pid:    3028
PPid:   2994
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
```

```
Utrace: 0
FDSize: 256
Groups: 1000
VmPeak:     3892 kB
VmSize:     3764 kB
VmLck:         0 kB
VmHWM:       380 kB
VmRSS:       380 kB
VmData:       48 kB
VmStk:        84 kB
VmExe:         4 kB
VmLib:      1548 kB
VmPTE:        32 kB
Threads:        1
SigQ:   2/16117
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: ffffffffffffffff
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed:
00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,0000
0000,00000000,00000000,00000000,00000000,00000000,00000000,00000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        2
nonvoluntary_ctxt_switches:     0
```

Note that the two processes have their own virtual address space, and there by multiply the memory usage.

Hands-on Lab

Problem: process with memory leakage.

In the below C program main() function waits for user input and then iteratively allocates memory from heap using malloc(). Each iteration allocates 8KB memory and doesnot free the allocated memory. This way memory gets leaked in each iteration. After a certain number of iterations, malloc() returns NULL, due to unavailability of memory. After receiving NULL, loop gets terminated and wait for user-input.

```
$ vi memleak.c

#include <stdio.h>
#include <stdlib.h>

int main()
{
  char ch;
  void *ptr;
  int i;
  printf("Before the loop\n");
  scanf("%c", &ch);
  for(i=1; ;i++) {
    printf("i=%d\n", i);
    ptr = malloc(8*1024); // 8KB
    if (ptr==NULL) {
      printf("malloc() failed\n");
      break;
    }
  }
  printf("After the loop\n");
  scanf("%c", &ch);
}
```

Run the above program in one terminal. The process starts and waits for user. From a second terminal, observe maps file of the process. This is the virtual address space before doing any dynamic memory allocations. Now enter any input in the first terminal. The process starts doing malloc() iteratively. After the process exhausts its virtual address space, it receives NULL from malloc(). It waits for user-input. Now in the second terminal, again observe maps of the process. You should see that virtual address space reaches near to 2GB. You should notice that heap segment becoming fat.

Problem: process with infinite recursion.

The below C program which has a user defined function func(). The function has a local variable of 1MB size. The function is a recursive function without any exit criteria. In other words, func() calls itself(). The main() waits for user input and then calls func().

```
$ vi recursion.c

#include <stdio.h>
void func(int i)
{
  printf("%d \n", i);
  func(i+1);
}
int main()
{
  char ch;
  printf("before calling func()\n");
  scanf("%c",&ch);
  func(0);
}
```

Run the above program in one terminal. The process starts and waits for user. From a second terminal, observe maps file of the process. This is the virtual address space before doing the function call to func(). Now enter any input in the first terminal. The process starts recursion infinitely. After the process exhausts its virtual address space, it receives segmentation violation. The process gets aborted by operating system. Unfortunately you cannot observe virtual space. You should notice that stack segment became fat.

Exercise:

1. Write a program which has a global array variable which occupies 200MB memory. Compile the program and run it. See the Virtual address space of the program. Now, comment the global variable definition and recompile the program. See the Virtual address space of the program. Notice the key differences between the address space segments.

2. In the above example we saw that size of stack grows when functions are called. We also saw that /proc/<pid>/maps file shows start and end virtual address of each segment. Write a program which contains function calls. Using the program demonstrate the order of growth of stack. In other words, find if stack grows towards lower addresses or higher addresses.

Threads

Thread:
A sequence of instructions executed within the context of a process.

Multithreading:
Multiple & independent sequences of instructions executed within the context of a process.

User-level or Application-level threads:
Threads managed by threads routines in user space, as opposed to kernel space. The POSIX pthread APIs are used to create and handle user threads.

Lightweight processes:
Kernel threads, also called LWPs, that execute kernel code and system calls.

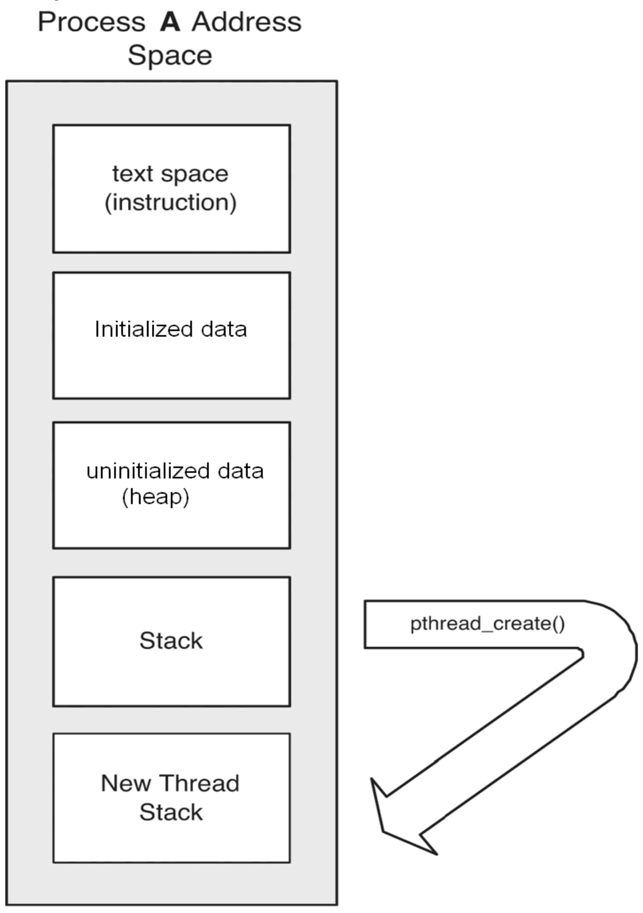
Thread: A sequence of instructions executed within the context of a process. or simply put thread is a “flow of control”. Every process contains at least one thread.

Multithreading: The word *multithreading* can be translated as *multiple threads of control* or *multiple flows of control*. While a traditional UNIX process contains a single thread of control, multithreading (MT) separates a process into many execution threads. Each of these threads run independently

User-level or Application-level threads
Threads managed by threads routines in user space, as opposed to kernel space. The POSIX pthreads APIs are used to create and handle user threads. In this manual, and in general, a thread is a user-level thread.
Note – Because this workshop is for application programming, kernel thread programming is not discussed.

Lightweight processes: Kernel threads, also called LWPs, that execute kernel code and system calls. LWPs are managed by the system thread scheduler, and cannot be directly controlled by the application programmer. In Linux 2.6 kernel, every user-level thread has a dedicated LWP. This is known as a 1:1 thread model.

Threads (contd…)



.

Text segment: Text segment contains program's binary code and libraries' binary code. This segment is per process. All threads in a process have one and the same text segment. In other words, all threads in a process share the same text segment.

Data segment: Data segment contains two parts - the global variables (also called Initialized data) and dynamically allocated variables (also called Uninitialized data or heap). All threads in a process have one and the same data segment. In other words, all threads in a process share data segment.

Stack segment: Stack contains Activation records (also called stack frames) of functions. All threads in a process have their own stack segment. When a new thread is created (using pthread_create()), a new stack segment is added to the process's virtual address space.

Virtual address space of multi-threaded process

Example

```
$  vi thr_sample.c

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void *thread_main(void *arg) {
  char *ret, ch;

  sleep(30);
  pthread_exit(NULL);
}

int main() {
  pthread_t thid;
  void *ret;
  char ch;

  printf("%s: Before pthread_create(). Press any key\n", __func__);
  scanf("%c", &ch);
  if (pthread_create(&thid, NULL, thread_main, NULL) != 0) {
    printf("pthread_create() error\n");
    exit(1);
  }
  printf("pthread_create() succeeded thid=%ld\n", thid);

  if (pthread_join(thid, &ret) != 0) {
    printf("pthread_join() error\n");
    exit(2);
  }
  printf("%s : After pthread_join(). Press any key\n", __func__);
  scanf("%c", &ch);
  free(ret);
}
```

Compile the program

```
$ gcc –o thr_sample –g –lpthread thr_sample.c
```

Execute the program in one terminal. But don’t press any key.

```
$ ./thr_sample
main: Before pthread_create(). Press any key
```

View threads in a process using “ps” command’s -L option from other terminal

```
$ ps -f  -L –U maruthisi
UID        PID  PPID  LWP C  NLWP STIME TTY  STAT TIME  CMD
…
maruthi+ 3765 2736  3765 0     1 06:55 pts/1 S+  0:00 ./thr_sample
…
```

Using the process id, see virtual address space of the process from other terminal.

```
$ cat /proc/3765/status
Name:   thr_sample
State:  S (sleeping)
Tgid:   3765
Ngid:   0
Pid:    3765
PPid:   2736
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
FDSize: 256
Groups: 1000
VmPeak:     6440 kB
VmSize:     6332 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:       700 kB
VmRSS:       700 kB
VmData:       76 kB
VmStk:       136 kB
VmExe:         4 kB
VmLib:      1976 kB
VmPTE:        32 kB
VmPMD:        12 kB
VmSwap:        0 kB
Threads:        1
SigQ:   0/14719
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000180000000
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 0000003fffffffff
Seccomp:        0
Cpus_allowed:   3f
Cpus_allowed_list:      0-5
Mems_allowed:
00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000
000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,0
0000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        1
nonvoluntary_ctxt_switches:     1
```

Now, in the first terminal, where program is running, press any key. At this point, new thread will be created.

```
$ ./thr_sample
main: Before pthread_create(). Press any key

pthread_create() succeeded thid=140075766335232
thread_main : Before pthread_exit(). Press any key
main : After pthread_join(). Press any key
```

After creating first thread, view threads in a process using “ps” command’s -L option from the other terminal.

```
$ ps -f  -L –U maruthisi
UID       PID  PPID  LWP C  NLWP STIME TTY  STAT TIME  CMD
…
maruthi+ 3765 2736  3765 0     1 06:55 pts/1 S+  0:00 ./thr_sample
maruthi+ 3765 2736  3778 0     1 06:55 pts/1 S+  0:00 ./thr_sample
…
```

Again, see virtual address space again from other terminal

```
$ cat /proc/3765/status
Name:   thr_sample
State:  S (sleeping)
Tgid:   3765
Ngid:   0
Pid:    3765
PPid:   2736
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
FDSize: 256
Groups: 1000
VmPeak:    14660 kB
VmSize:    14660 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:       700 kB
VmRSS:       700 kB
VmData:     8404 kB
VmStk:       136 kB
VmExe:         4 kB
VmLib:      1976 kB
VmPTE:        36 kB
VmPMD:        12 kB
VmSwap:        0 kB
Threads:        2
SigQ:   0/14719
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000180000000
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 0000003fffffffff
Seccomp:        0
Cpus_allowed:   3f
Cpus_allowed_list:      0-5
Mems_allowed:
00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000
000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,0
0000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        2
nonvoluntary_ctxt_switches:     1
```

Another way to view threads of a process is to list the /proc file system's task directory /proc/<pid>/task/

```
$ ls -l /proc/3677/task/
total 0
dr-xr-xr-x. 7 maruthisi maruthisi 0 Jun 17 06:54 3677
dr-xr-xr-x. 7 maruthisi maruthisi 0 Jun 17 06:54 3742
```

Advantages

Multithreading your code can

- Improve application responsiveness
- Use multiprocessors more efficiently
- Improve program structure
- Use fewer system resources

Multi-threaded programming is easier than multi-process programming.

Improving Application Responsiveness
Any program in which many activities are not dependent upon each other can be redesigned so that each independent activity is defined as a thread.

Using Multiprocessors Efficiently
Typically, applications that express concurrency requirements with threads need not take into account the number of available processors. The performance of the application improves transparently with additional processors because the operating system takes care of scheduling threads for the number of processors that are available. When multicore processors and multithreaded (aka hyper-threaded) processors are available, a multithreaded application's performance scales appropriately because the cores and threads are viewed by the OS as processors.

Improving Program Structure
Many programs are more efficiently structured as multiple independent or semi-independent units of execution instead of as a single, monolithic thread. For example, a non-threaded program that performs many different tasks might need to devote much of its code just to coordinating the tasks. When the tasks are programmed as threads, the code can be simplified. Multithreaded programs, especially programs that provide service to multiple concurrent users, can be more adaptive to variations in user demands than single-threaded programs.

Using Fewer System Resources
Programs that use two or more processes that access common data through shared memory are applying more than one thread of control. However, each process has a full address space and operating environment state. Cost of creating and maintaining this large amount of state information makes each process much more expensive than a thread in both time and space. In addition, the inherent separation between processes can require a major effort by the programmer. This effort includes handling communication between the threads in different processes, or synchronizing their actions. When the threads are in the same process, communication and synchronization becomes much easier.

If you're a disciplined programmer, designing and coding a multithreaded program should be easier than designing and coding a multiprocess program.

Multi-threaded programming support in Linux

Two sets of API:

- POSIX pthreads (also called pthreads) or NPTL threads
- LinuxThreads

Libraries:

- libpthread.so
- libthread.so

Differences between POSIX pthreads and Linux threads:

- POSIX threads are more portable.
- POSIX threads establish characteristics for each thread according to configurable attribute objects.
- POSIX pthreads implement thread cancellation.
- POSIX pthreads enforce scheduling algorithms.
- POSIX pthreads allow for clean-up handlers for fork(2) calls.
- Linux threads can be suspended and continued.
- Linux threads don't interporate well with signals.

Linux, provides an implementation of POSIX pthread specification, which is called Native POSIX Thread Library (NPTL). It is provided as a separate library viz., libpthread. POSIX threads are guaranteed to be fully portable to other POSIX-compliant environments (Linux, Solaris, HP-UX, AIX, other Unices).

Linux also provides other threading implementation. Viz., Linux Threads. After Linux 2.6 kernel, the Linux Threads library is deprecated.

This workshop only covers POSIX pthreads and doesn't cover the other implementation.

Guidelines for multi-threaded programming

Some guidelines for multi-threaded programming

- Always write re-entrant code (also called thread safe programs)
- Make sure that routines are also re-entrant.
- When accessing global variables or dynamically allocated memory use proper synchronization primitives.
- don't use bigger local variables
- When developing libraries make sure they are thread safe
- Don't link your mulit-threaded programs with thread-unsafe libraries.
- Don't use interactive functions (eg scanf()) in more than one threads.

Some guidelines for re-entrancy of routines:

- Must hold no static (global) non-constant data.
- Must not return the address to static (global) non-constant data.
- Must work only on the data provided to it by the caller.
- Must not rely on locks to singleton resources.
- Must not modify its own code -- no self-modifying code
- Must not call non-reentrant routines.

Thread safe libraries:
A library in which all routines follow the above guidelines.

Write re-entrant code. (aka thread safe program) : A routine is described as **reentrant** if it can be safely executed concurrently; that is, the routine can be re-entered while it is already running.

The below example, function f() is not a re-entrant routine, and hence g() is also not.

```
int g_var = 1;

int f()
{
  g_var = g_var + 2;
  return g_var;
}

int g()
{
  return f() + 2;
}
```

Hands-on Lab

Problem: In a Linux machine, try to locate POSIX pthread library

```
$ ls -l /usr/lib/libpthread.*
lrwxrwxrwx. 1 root root      18 Feb 23  2015 /usr/lib/libpthread.so.0
-> libpthread-2.21.so
```

Problem: In the Linux, see if there are any programs running (processes) which are mult-threaded.

Use –L option of ps to list light weight process ID (LWP column), and number of LWPs (NWLP column) and redirect to a file.

```
$ ps –ef –L > allproc.txt
```

Open the file in your preferred editor

```
$ vi allproc.txt
```

See if the NLWP column has a number other than 1. All such processes are multi-threaded.

Problem: See virtual address space of any multi-threaded process found in the above experiment.

Use the pid of the multi-threaded process found in above experiment, and see maps file and study the virtual address space. You might receive permission denied, if you don't have appropriate permissions. For eg, trying to see virtual address space of root's process, when you are logged in as a normal user.