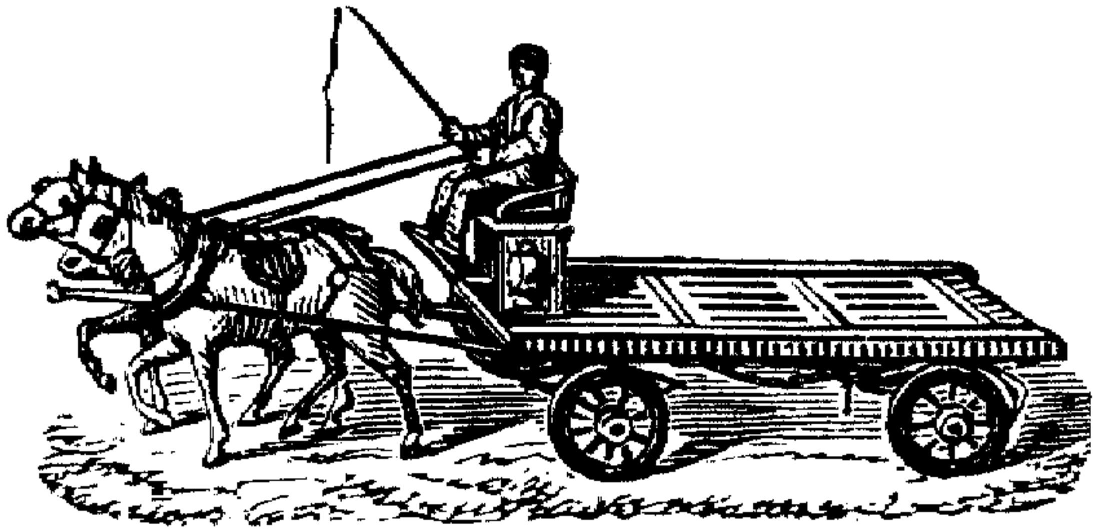
Workshop on

Multi-threaded programming in C



by

Maruthi Seshidhar Inukonda

Document revision history:

2007: “Workshop on Programming for Symmetric Multi-Processors” made for solaris.
2010: “Workshop on Programming for Symmetric Multi-Processors” ported to Linux.
2016 June: Renamed as “Multi-threaded programming in C”. Added CC BY-NC-SA license.

Preface

A decade back processor manufacturers used to produce processors with increasing clock speed. (1.8 GHz, 2.4 GHz, 3.0 GHz). From last few years, processor manufacturers have started producing multi CPU products (Multi-threaded, Dual Core, Quad Core, Oct-Core).

Buying a multi CPU processor doesn't boost performance of a software automatically, Software should be developed to exploit the parallelism provided by these architectures. Multi-Threaded programming paradigm is gaining importance with these contemporary processors.

This workshop is aimed at pthread programming, which is most commonly used for concurrent programming in Unix world. It starts with survey of computers that have multi CPU systems (Module 1), and describes the reduction of memory foot print due to multi-threaded processes (Module 2, Module 3). Pthreads programming is covered next. (Module 4, Module 5). At last touches on Design patterns for Multi-threaded programs (Module 6). All modules are supported by examples. Entire content for the workshop is customized for Linux. Since programs using pthreads API are portable, the examples should work with no change in other Unix (Solaris, HP-UX, AIX)

Please let me know if you have any feedback on the workshop / workshop material at my mail-id.

Acknowledgements

During my under graduation, I got great help from my computer sciences lecturer, Rama Krishna Sunkara, who is a good programmer. He used to inspire me 1%, and at the same time used to keep me perspire 99% by giving lots and lots of daily programming exercises. I am very much grateful to him. From those days, I got motivated to help students in their programming. So I have made a hobby to occasionally take-up programming related courses, workshops and lectures.

Maruthi Seshidhar Inukonda,
M.Tech(CSE) IIT Roorkee,
maruthi.inukonda [at] gmail.com

Contents

Module 1 : Introduction

Trends in Processors

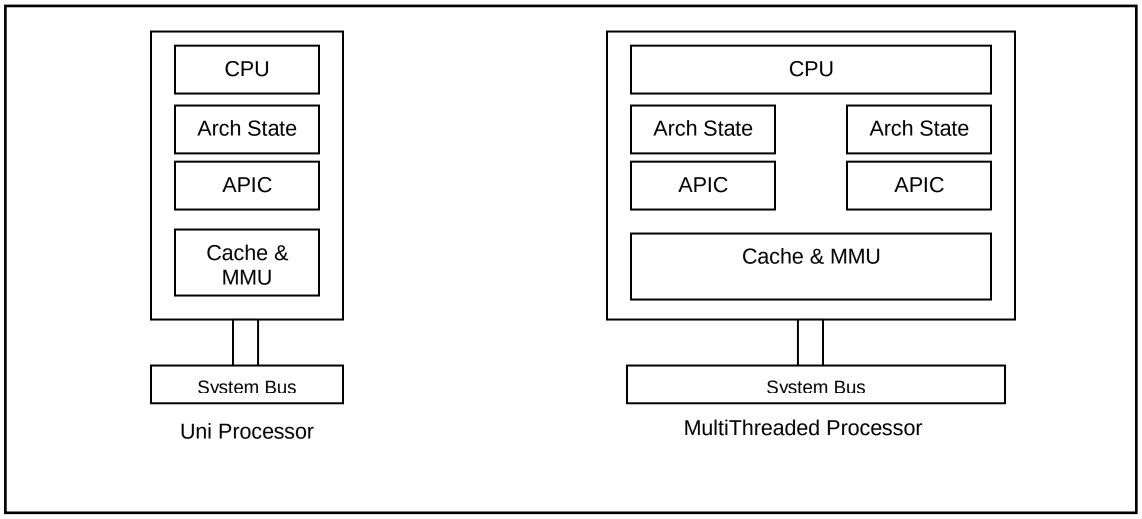
In the arena of Desktop/Workstation computers,

- only one processor per computer till 2003.
- till 2003, rapid increase in processor clock speed (1.8, 2.4, 3.0 GHz)
- In 2003, concept of multi-threading in processors was introduced
- In 2004, concept of multi-core processors was introduced
- Currently the focus is in increasing number of CPU in one silicon die. (Dual-Core, Quad-Core, Octa-Core).

In the arena of Server computers,

- More than one processor per computer till 2003
- Before 2003, increase in number of processors per computer, and also increase in processor clock speed (1.5 GHz, 1.8 GHz)
- In 2003, concept of multi-threading was borrowed from Desktop processors
- In 2004, concept of multi-core processors was also borrowed
- Currently the focus is in increasing number of CPU in one silicon die, along with number of processors in one computer.

Terminology



.

Uni Processor system:

- A computer system with a single "Central Processing Unit" [*] on a single silicon die.
- One Architecture State [†] and one APIC [‡] exists for the CPU.
- The CPU has its own Cache & MMU
- The CPU has its own access to system bus
- OS provide illusion of multi-tasking by time slicing processes (aka Time Sharing).
- Eg: Intel's 80x86, Pentium III, Pentium IV

Multi Threaded System:

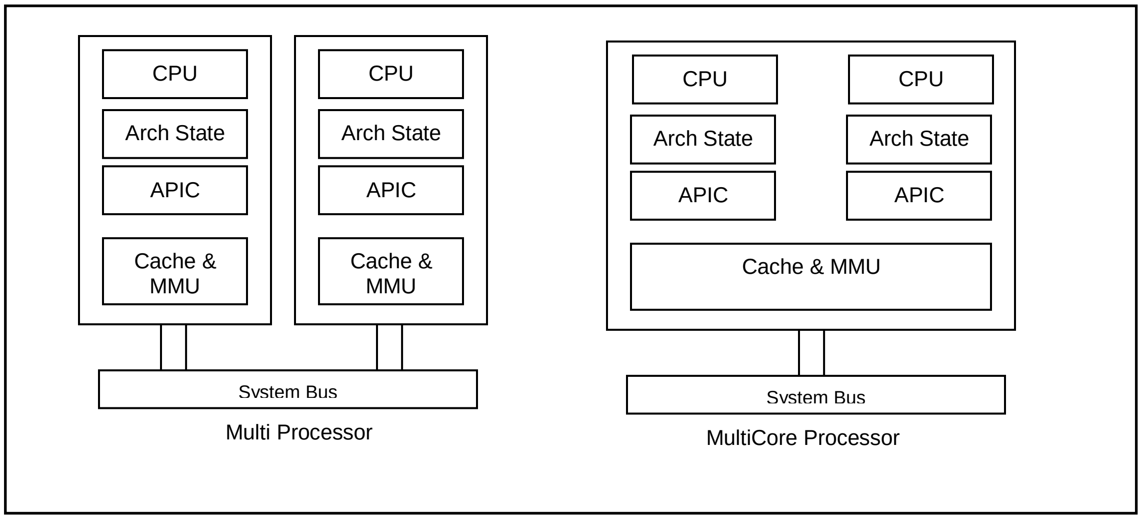
- A computer system with one CPU in a single silicon die.
- Multiple Architecture State and APIC are added in a traditional processor to enable one physical processor appear as multiple logical processors to the OS.
- The CPU has its own Cache & MMU
- The CPU has its own access to system bus
- OS provides multi-tasking by exploiting the pseudo parallelism.
- Eg. Intel's Pentium 4 *HT*, Xeon.

[*] People commonly & mistakenly use term CPU to refer to Cabinet/Casing. But here CPU refers to the electronic circuit that can execute computer program

[†] Architecture State contains Instruction Flag registers, MMU registers, Interrupt mask registers, Status registers, General purpose registers.

[‡] Advanced Programmable interrupt controller (APIC) is a device which allows priority levels to be assigned to its interrupt outputs. Intel's 8259A is well known APIC.

Terminology



Multi Processor system:

- A computer system with two or more CPUs each in a separate silicon die.
- Architecture State and APIC exist per CPU
- Each CPU has its own Cache and MMU.
- Each CPU has its own access to system bus
- OS provides multi-tasking by exploiting the true parallelism.
- Eg: Intel's Itanium, Sun's SPARC, IBM's Power 5.

Multi Core System:

- A computer system with two or more CPU in a single silicon die.
- Architecture State and APIC exist per CPU
- All CPUs in a core share Cache & MMU.
- All CPU in a core share access to system bus
- OS provides multi-tasking by exploiting the true parallelism.
- Eg. Intel's core2 Duo, Quad Core, Oct Core, Motecito.

Symmetric Multiprocessor (SMP) system:

- contains multi processor, where all processors are of the same kind
- SMP systems have no master or slave processors
- an SMP operating system assigns any task to any processors
- SMP programming is much simpler

Asymmetric Multiprocessor (ASMP) system:

- contains multi processor, where each processor is a special kind
- ASMP systems have master and slave processors
- an ASMP operating system assigns certain tasks only to certain processors
- ASMP programming is relatively difficult

Symmetric MultiThreaded (SMT) system:

- contains multi-threaded processor, where all logical processors are of the same kind
- SMT systems have no master or slave processors
- an SMT operating system assigns any task to any processors
- SMT programming is much simpler

Operating Systems supporting Multi-tasking.

On SMP machines, Operating System provides two features for applications:
Multi-Processing: Ability to run more than one process at the same time. Each processor runs one process at a time.

Multi-Threading: Ability to run more than one thread at the same time. Each processor runs one thread of a process at a time.

Many Operating Systems provide both the features:

- HP-UX
- Sun Solaris
- IBM AIX
- Linux (all distributions)
- Microsoft Windows Server, Windows XP
- Apple Macintosh

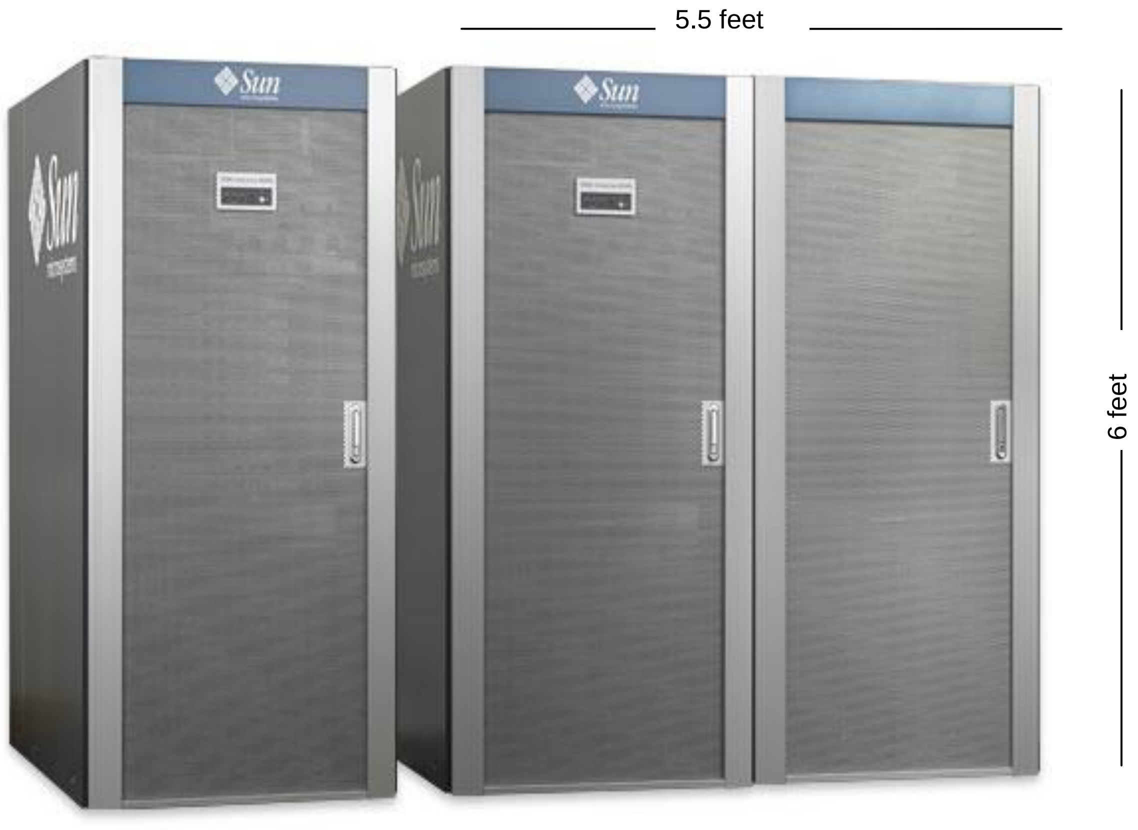
Example SMP Machines

Hewlett-Packard’s - Integrity Superdome server

Processor Type	Intel Itanium 2 – 1.6 GHz
No of Processors	64
No of Cores per Processor	4
RAM	2TB
Operating System	HP-UX 11iv3
Weight	1,200 Kg
Cost[§]:	31,500,000 INR

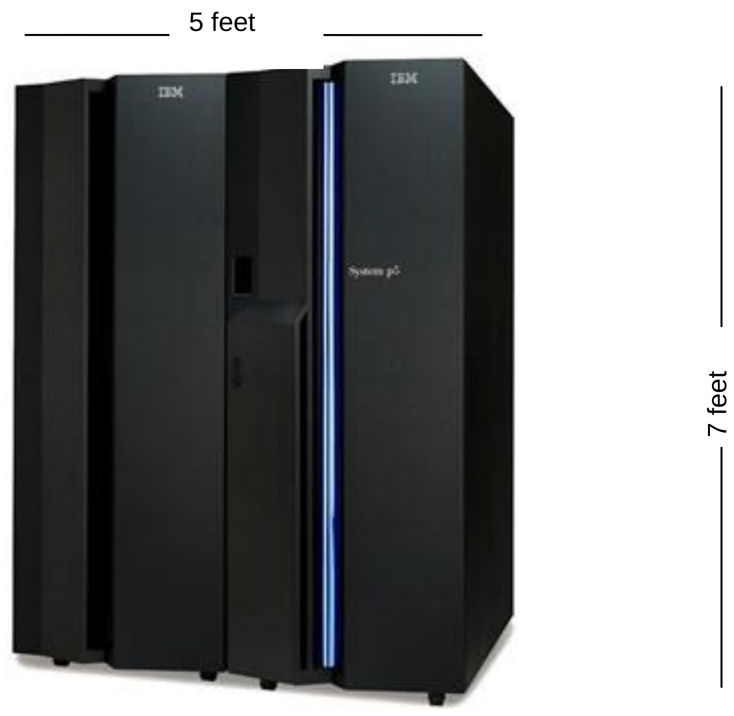
[§] Cost is approximate. It is just shown for Information, not for any other use.

Sun Microsystems's – Sun M9000 Server



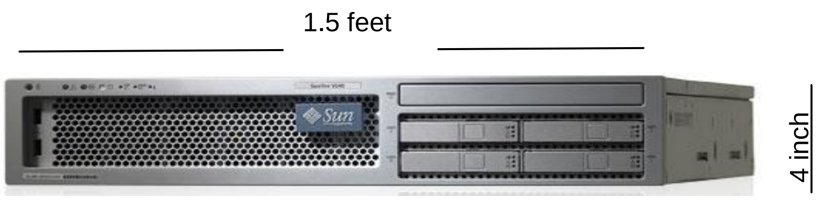
Processor Type	Sun SPARC VII – 1.6 GHz
No of Processors	64
No of Cores per Processor	4
RAM	2TB
Operating System	Solaris 10
Weight	1,880 Kg
Cost	28,305,000 INR

International Business Machines - p5 595 Unix Server



Processor Type	Power 5 – 2.3 GHz
No of Processors	4
No of Cores per Processor	64
RAM	2TB
Operating System	AIX 5L
Weight	2458 Kg
Cost	26,500,000 INR

Sun Microsystems's – Sun M9000 Server



Processor Type	Sun Ultra SPARC 1.5 GHz
No of Processors	2
No of Cores per Processor	1
RAM	16 GB
Operating System	Solaris 10
Weight	23 kg
Cost	180,000 INR

Desktop

Processor Type	Intel Core i5 – 2.33 GHz
No of Processors	1
No of Cores per Processor	4
RAM	8GB
Operating System	Windows, Linux, Solaris X86
Weight	10 kg
Cost	40,000 INR

Laptop

Processor Type	Intel Core i3 – 2.8 GHz
No of Processors	1
No of Cores per Processor	2
RAM	8GB
Operating System	Windows, Linux
Weight	3kg
Cost	45,000 INR

Hands-on Lab

Problem 1: Find number of processors using Linux.

To see number of physical processors use run the following command in a terminal.

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Core(TM)2 Duo CPU     T7300  @ 2.00GHz
stepping        : 10
cpu MHz         : 800.000
cache size      : 4096 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov
                  pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
                  syscall nx lm constant_tsc arch_perfmon pebs bts rep_good
                  pni dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm
                  lahf_lm ida tpr_shadow vnmi flexpriority
bogomips        : 3989.86
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Core(TM)2 Duo CPU     T7300  @ 2.00GHz
stepping        : 10
cpu MHz         : 800.000
cache size      : 4096 KB
physical id     : 0
siblings        : 2
core id         : 1
cpu cores       : 2
apicid          : 1
initial apicid  : 1
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov
                  pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
                  syscallnx lm constant_tsc arch_perfmon pebs bts rep_good
                  pni dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm
                  lahf_lm ida tpr_shadow vnmi flexpriority
bogomips        : 3989.80
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

Module 2 : Processes

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
```

In this case compiler internally calls linker.

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

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/

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.

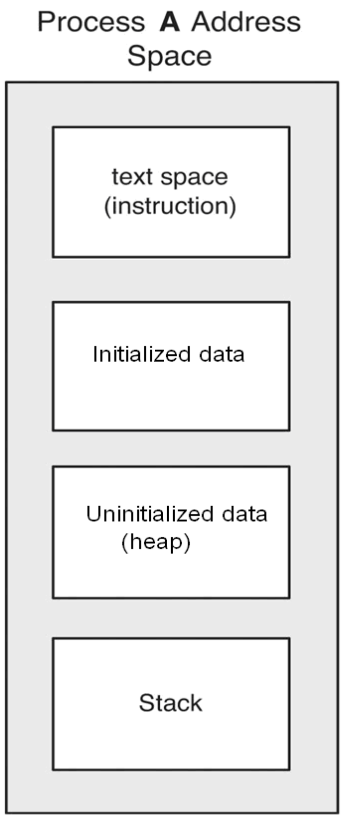
Address Space:

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…)

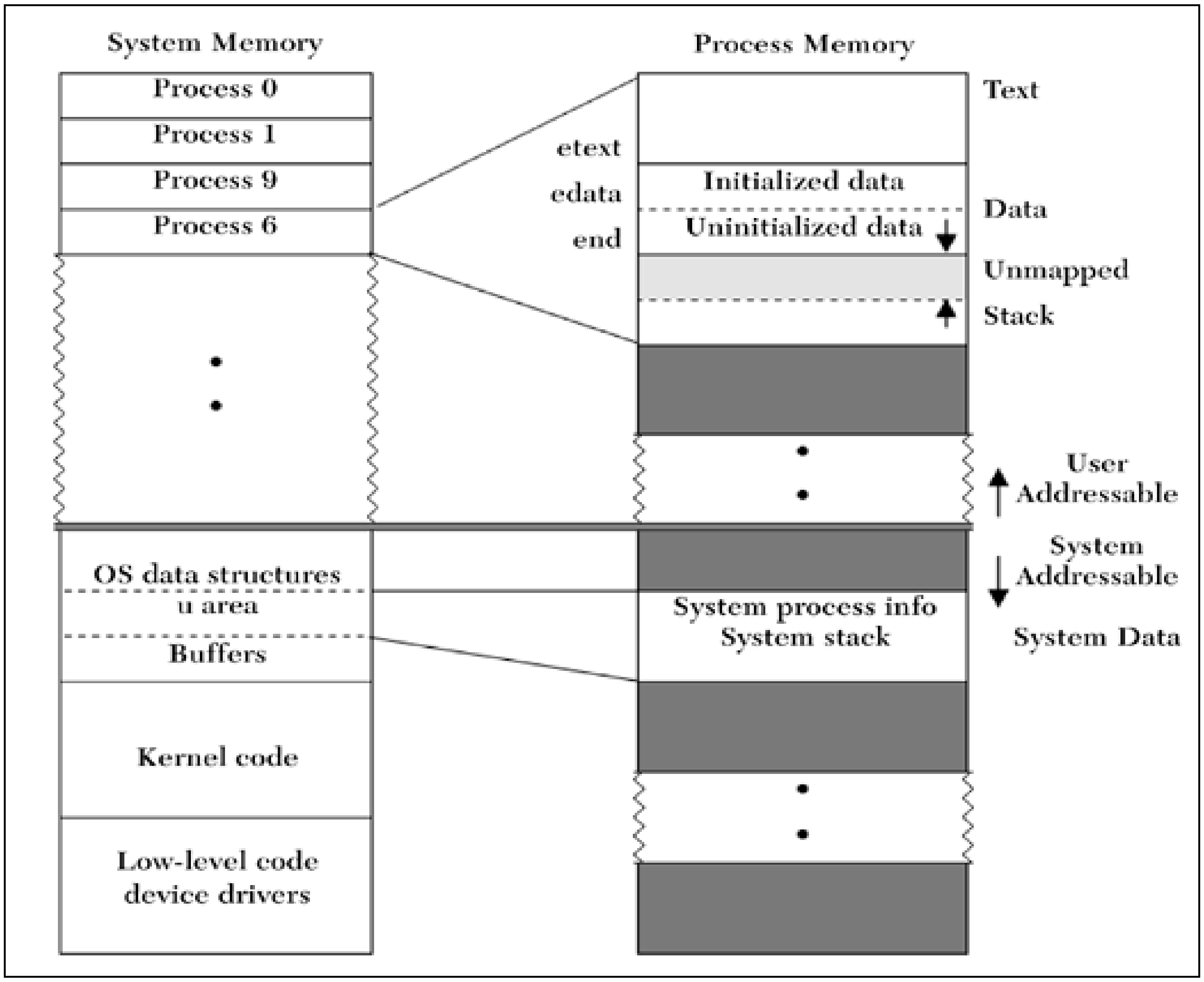


Figure above shows system memory, containing processes in user space and OS data structures, u-area, buffers, kernel code, device drivers code in kernel space. Figure also shows zoom-in of a process's virtual address space.

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:    19774   19774   19774   19774
Gid:    30      30      30      30
Utrace: 0
FDSize: 256
Groups: 30
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:    19774   19774   19774   19774
Gid:    30      30      30      30
Utrace: 0
FDSize: 256
Groups: 30
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:    19774   19774   19774   19774
Gid:    30      30      30      30
Utrace: 0
FDSize: 256
Groups: 30
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.

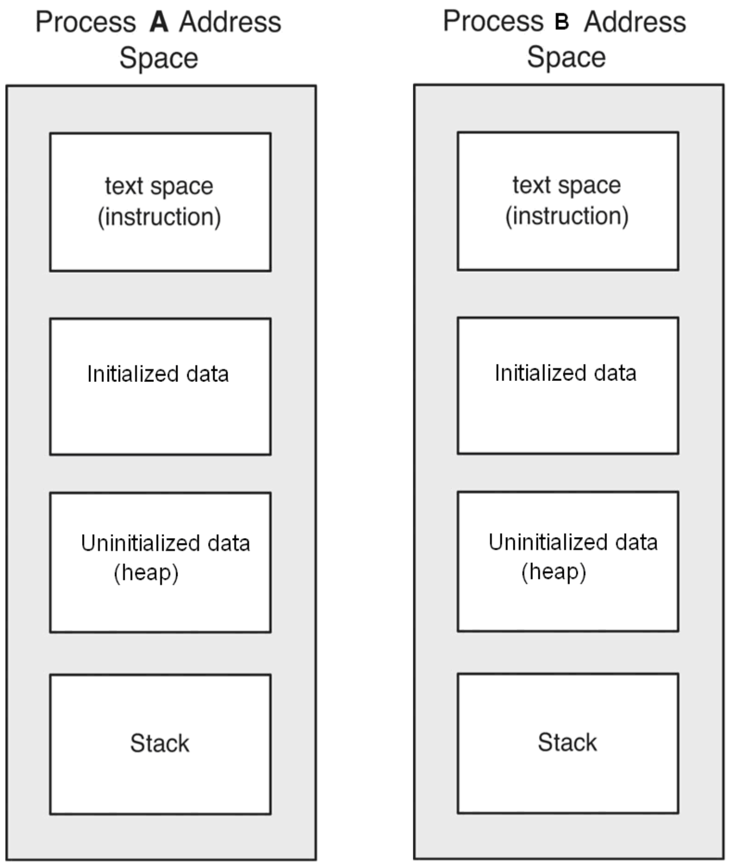
```
$ cat /proc/2936/status
Name:   stack
State:  S (sleeping)
Tgid:   2936
Pid:    2936
PPid:   2739
TracerPid:      0
Uid:    19774   19774   19774   19774
Gid:    30      30      30      30
Utrace: 0
FDSize: 256
Groups: 30
VmPeak:     3892 kB
VmSize:     3760 kB
VmLck:         0 kB
VmHWM:       368 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:    19774   19774   19774   19774
Gid:    30      30      30      30
Utrace: 0
FDSize: 256
Groups: 30
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:    19774   19774   19774   19774
Gid:    30      30      30      30
Utrace: 0
FDSize: 256
Groups: 30
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:    19774   19774   19774   19774
```

```
Gid:    30      30      30      30
Utrace: 0
FDSize: 256
Groups: 30
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 pmap 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 pmap 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 pmap 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.

Module 3 : Threads

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 pthreads and Solaris threads APIs are used to create and handle user threads.

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

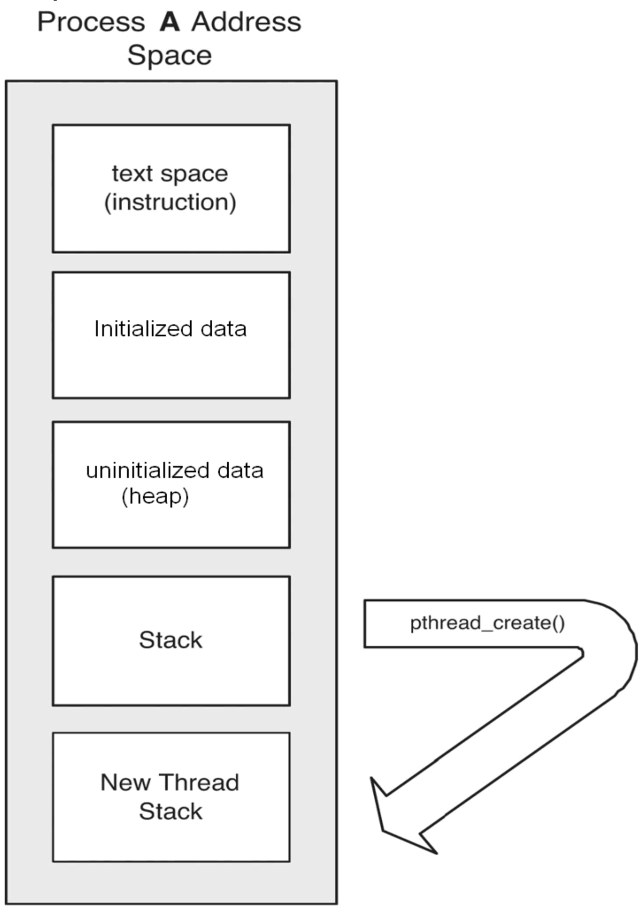
Thread: A sequence of instructions executed within the context of a process. or simply put thread is a “flow of control”. Every process contains at least one thread.

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 and Solaris threads 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:    1001    1001    1001    1001
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:
00000000,00000000,00000000,00000000,00000000,00000000,00000000,000000
00,00000000,00000000,00000000,00000000,00000000,00000000,00000000,000
00001
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:    1001    1001    1001    1001
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:
00000000,00000000,00000000,00000000,00000000,00000000,00000000,000000
00,00000000,00000000,00000000,00000000,00000000,00000000,00000000,000
00001
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 fam 0 Jun 17 06:54 3677
dr-xr-xr-x. 7 maruthisi fam 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 (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 run pmap command 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.

Module 4 *:* Pthreads Programming

Thread Creation

Prototype:
#include <pthread.h>

*int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr,
void *(*start_routine)(void*), void *restrict arg);*

Parameters:

Parameter	Direction	Description
thread	out	if thread creation is successful, *thread* contains thread id. Must not be NULL.
attr	in	thread attributes. May be NULL, in which case, default attributes are used.
start_routine	in	first function to be called immediately after creating the thread. function signature should be void* myfunc(void*).
arg	in	the argument to be passed to *start_routine()*

Return Value:
If successful, the *pthread_create()* function returns 0. Otherwise, an error number is returned to indicate the error.

Description:
The pthread_create() function is used to create a new thread, with attributes specified by attr, within a process. If attr is NULL, the default attributes are used. Attributes are used to specify non-default scheduling policy, non-default stack size, etc. Upon successful completion, pthread_create() stores the ID of the created thread in the location referenced by thread. The thread is created executing start_routine with arg as its sole argument.

Example:

```
$ vi thr_create.c

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

void *thread_main(void *arg) {
  printf("%s: argument '%s'\n", __func__, arg);

  sleep(60);

  printf("%s : Before returning.\n", __func__);
  return NULL;
}

int main() {
  pthread_t thid;
  char ch;

  printf("%s : Before create. Press any key\n", __func__);
  scanf("%c", &ch);

  if (pthread_create(&thid, NULL, thread_main, "argument1") != 0) {
    printf("pthread_create() error\n");
    exit(1);
  }
  printf("%s: create succeeded thid=%d\n", __func__, thid);

  printf("%s : Before returning. Press any key\n", __func__);
  scanf("%c", &ch);
}
```

Compile the program.

```
$ gcc -o thr_create -lpthread thr_create.c
```

Run the program in one terminal, and when the program waits at the first scanf() in main(), don’t press any key.

```
$ ./thr_create
main : Before create. Press any key
```

Open second terminal and see threads of the thr_create process using ps command

```
$ ps -f -L -U maruthisi
UID       PID  PPID  LWP C NLWP STIME TTY       TIME CMD
maruthi+ 2031 1786 2031 0    1 06:50 pts/ 00:00:00 ./thr_create
```

Now in the first terminal, enter any key. The main() creates a thread and again waits at the second scanf(). Don’t enter any key. Where as the new thread starts executing thread_main() and sleeps for 60 secs.

```
$ ./thr_create
main : Before create. Press any key

main: create succeeded thid=-1299540224
thread_main: argument 'argument1'
main : Before returning. Press any key
thread_main : Before returning.
```

Now in the second terminal, again see threads of the thr_create process using ps command.

```
$ ps -f -L -U maruthisi
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
maruthi+ 2031 1786 2031 0    1 06:50 pts/  00:00:00 ./thr_create
maruthi+ 2031 1786 2054 0    2 06:55 pts/1 00:00:00 ./thr_create
```

Now in the first terminal after 60 seconds, new thread returns. After the new thread returns, enter any key. The main() function terminates.

Thread Identification

Prototype:
#include <pthread.h>

    pthread_t pthread_self(void);

Parameters:
None

Return Value:
returns the thread ID of the calling thread.

Description:
The pthread_self() function returns the thread ID of the calling thread. This thread ID should be same as the ID returned to the calling thread's creator when pthread_create() was called.

Example:

```
$ vi thr_self.c

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

void *thread_main(void *arg) {
  printf("%s:%u argument '%s'\n", __func__, pthread_self(), arg);

  sleep(60);

  printf("%s:%u Before returning.\n", __func__, pthread_self());
  return NULL;
}

int main() {
  pthread_t thid;
  char ch;

  printf("%s:%u   Before   create.   Press   any   key\n",   __func__,
pthread_self());
  scanf("%c", &ch);

  if (pthread_create(&thid, NULL, thread_main, "argument1") != 0) {
    printf("pthread_create() error\n");
    exit(1);
  }
  printf("%s: create succeeded thid=%d\n", __func__, thid);

  printf("%s:%u  Before  returning.  Press  any  key\n",  __func__,
pthread_self());
  scanf("%c", &ch);
}
```

Compile the program.

```
$ gcc -o thr_self -lpthread thr_self.c
```

Run the program

```
$ ./thr_self
main:2415888128 Before create. Press any key

main: create succeeded thid=-1887324416
main:2415888128 Before returning. Press any key
thread_main:2407642880 argument 'argument1'
```

Thread Destruction

Prototype:
#include <pthread.h>

*void pthread_exit(void *value_ptr);*

Parameters:

Parameter	Direction	Description
value_ptr	in	pointer to a memory location which should be made available to the parent thread calling pthread_join(). May be NULL.

Return Value:
The pthread_exit() function cannot return to its caller.

Description:
The pthread_exit() function terminates the calling thread, in a similar way that exit() terminates the calling process. If the thread is not detached, the exit status specified by value_ptr is made available to any successful join with the terminating thread.

After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. Thus, references to local variables of the exiting thread should not be used for the pthread_exit() value_ptr parameter value.

Example:

```
$ vi thr_exit.c

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

void *thread_main(void *arg) {
  printf("%s: argument '%s'\n", __func__, arg);

  sleep(60);

  printf("%s : Before exiting thread\n", __func__);
  pthread_exit(NULL);
}

int main() {
  pthread_t thid;
  char ch;

  printf("%s : Before create. Press any key\n", __func__);
  scanf("%c", &ch);

  if (pthread_create(&thid, NULL, thread_main, "argument1") != 0) {
    printf("pthread_create() error\n");
    exit(1);
  }
  printf("%s: create succeeded thid=%u\n", __func__, thid);

  printf("%s : Before returning. Press any key\n", __func__);
  scanf("%c", &ch);
}
```

Compile the program.

```
$  gcc -o thr_exit -lpthread thr_exit.c
```

Run the program

```
$ ./thr_exit
main : Before create. Press any key

main: create succeeded thid=4256118528
main : Before returning. Press any key
thread_main: argument 'argument1'
```

Thread Join

Prototype:
#include <pthread.h>

*int pthread_join(pthread_t thread, void **status);*

Parameters:

Parameter	Direction	Description
thread	in	thread id of the thread for which the calling thread should block.
status	out	memory location in which the *value_ptr* pointer passed to pthread_exit(), should be placed.

Return Value:
If successful, pthread_join() returns 0. Otherwise, an error number is returned to indicate the error.

Description:
The pthread_join() function suspends processing of the calling thread until the target thread completes. *thread* must be a member of the current process and it cannot be a detached thread. The pthread_join() function will not block processing of the calling thread if the target thread has already terminated.

If a pthread_join() call returns successfully with a non-null status argument, the value passed to pthread_exit() by the terminating thread will be placed in the location referenced by status.

Example:

```
$ vi thr_join.c

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

void *thread_main(void *arg) {
  char *ret;
  printf("%s: argument '%s'\n", __func__, arg);

  sleep(60);

  ret = (char*) malloc(20);
  if (ret == NULL) {
    printf("malloc() error\n");
    exit(2);
  }
  strcpy(ret, "This is a test");

  printf("%s : Before exiting thread\n", __func__);
  pthread_exit(ret);
}

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

  printf("%s : Before create. Press any key\n", __func__);
  scanf("%c", &ch);

  if (pthread_create(&thid, NULL, thread_main, "argument1") != 0) {
    printf("pthread_create() error\n");
    exit(1);
  }
  printf("%s: create succeeded thid=%u\n", __func__, thid);

  if (pthread_join(thid, &ret) != 0) {
    printf("pthread_join() error\n");
    exit(3);
  }

  printf("thread exited with '%s'\n", ret);
  printf("%s : After pthread_join(). Press any key\n", __func__);
  scanf("%c", &ch);
  free(ret);

}
```

Compile the program.

```
$  gcc -o thr_join -lpthread thr_join.c
```

Run the program

```
$ ./thr_join
main : Before create. Press any key

main: create succeeded thid=559707904
thread_main: argument 'argument1'
thread_main : Before exiting thread
thread exited with 'This is a test'
main : After pthread_join(). Press any key
```

Hands-on Lab

Problem: Increment each value in a given array

The below program defines a structure vector to represent a Vector. A function incr() which walks through the vector and increments each of the element. This program is a single-threaded version.

```
$ vi incr.c

#include<stdio.h>

typedef struct {
  int *ar;
  long n;
} vector;


void *
incr(void *arg)
{
  long i;

  for (i = 0; i < ((vector *)arg)->n; i++)
      ((vector *)arg)->ar[i]++;
}


int     ar[10000000];

int main(void)
{
  vector   vec;

  vec.ar = &ar[0];
  vec.n  = 10000000;

  incr(&vec);

  return 0;
}
```

Compile the program

```
$ gcc -o incr incr.c
```

Run the program & time it.

```
$ time ./incr

real    0m0.072s
user    0m0.050s
sys     0m0.023s
```

Now use the mult-threaded version of the same program.

```
$ vi thr_incr.c

#include<stdio.h>
#include<pthread.h>

typedef struct {
  int *ar;
  long n;
} vector;


void *
incr(void *arg)
{
  long i;

  for (i = 0; i < ((vector *)arg)->n; i++)
      ((vector *)arg)->ar[i]++;
}

int        ar[10000000];

int main(void)
{
  pthread_t  th1, th2;
  vector   vec1, vec2;


  vec1.ar = &ar[0];
  vec1.n  = 5000000;
  (void) pthread_create(&th1, NULL, incr, &vec1);


  vec2.ar = &ar[5000000];
  vec2.n  = 5000000;
  (void) pthread_create(&th2, NULL, incr, &vec2);


  (void) pthread_join(th1, NULL);
  (void) pthread_join(th2, NULL);
  return 0;
}
```

Compile the program

```
$ gcc -o thr_incr -lpthread thr_incr.c
```

Run the program & time it.

```
$ time ./thr_incr

real    0m0.060s
user    0m0.066s
sys     0m0.039s
```

You can notice that there is little benefit by multi-threading. See next problem where there is lot of performance boost.

Problem : Selection Sort

The below program creates a vector of size 1,00,000 entries, initializes it using random number generator rand(). It then calls sort() function, which does selection sort. The selection sort proceeds by iterating on the vector, and in each iteration (on index variable i) picks up the least element for that position. Note that this a single threaded version.

```
$ vi sort.c

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

typedef struct {
  int *ar;
  long beg;
  long end;
} vector;

#define SIZE 100000

void *
sort(void *arg)
{
  long i, j;
  long beg, end;
  int *ar;
  ar  = ((vector *)arg)->ar;
  beg = ((vector*)arg)->beg;
  end = ((vector *)arg)->end;
  for (i = beg; i <= end-1; i++) {
    for (j = i+1; j <= end; j++) {
      //printf("\n ar[%ld]=%d ar[%ld]=%d. ", i, ar[i], j, ar[j]);
      if (ar[i] > ar[j]) {
        int temp;
        temp = ar[i]; ar[i] = ar[j]; ar[j] = temp;
      }
    }
  }
}

int main(void)
{
  int      *ar = malloc(sizeof(int)*SIZE);
  vector   vec;
  long i;

  vec.ar   = &ar[0];
  vec.beg  = 0;
  vec.end  = SIZE-1;
  (void)srand(12345);

  for(i=vec.beg; i<=vec.end; i++) {
    vec.ar[i] = rand();
    //printf("%d\n", vec.ar[i]);
  }

  sort(&vec);

  //for(i=vec.beg; i<=vec.end; i++) {
    //printf("%d\n", vec.ar[i]);
  //}

  free(ar);
  return 0;
}
```

Compile the program

```
$ gcc -o sort sort.c
```

Run the program & time it.

```
$ time ./sort

real    0m41.952s
user    0m41.924s
sys     0m0.048s
```

Now use the multi-threaded version of the program, which splits the sorting job into two tasks each task is to sort 50,000 elements. The main thread creates two threads and gets the two tasks run in these two thread. The main thread waits till both of them complete. After that it merges the results to make a sorted array of 1,00,000 elements.

```
$ vi thr_sort.c

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

typedef struct {
  int *ar;
  long beg;
  long end;
} vector;

#define SIZE 100000

void *
sort(void *arg)
{
  long i, j;
  long beg, end;
  int *ar;
  ar  = ((vector *)arg)->ar;
  beg = ((vector*)arg)->beg;
  end = ((vector *)arg)->end;
  for (i = beg; i <= end-1; i++) {
    for (j = i+1; j <= end; j++) {
      //printf("\n ar[%ld]=%d ar[%ld]=%d. ", i, ar[i], j, ar[j]);
      if (ar[i] > ar[j]) {
        int temp;
        temp = ar[i]; ar[i] = ar[j]; ar[j] = temp;
      }
    }
  }
}

int main(void)
{
  int        *ar = malloc(sizeof(int)*SIZE);
  int        res[SIZE];
  long i,j,k;
  pthread_t  th1, th2;
  vector   vec1, vec2;

  (void)srand(12345);

  for(i=0; i<=SIZE-1; i++) {
    ar[i] = rand();
    //printf("%d\n", ar[i]);
  }

  vec1.ar = &ar[0];
  vec1.beg  = 0;
  vec1.end  = SIZE/2-1;
  (void) pthread_create(&th1, NULL, sort, &vec1);

  vec2.ar = &ar[0];
  vec2.beg  = SIZE/2;
  vec2.end  = SIZE-1;
  (void) pthread_create(&th2, NULL, sort, &vec2);

  (void) pthread_join(th1, NULL);

  (void) pthread_join(th2, NULL);

  // Merge the results
  i=0; j=SIZE/2; k=0;
  while ((i<=SIZE/2-1) && (j<=SIZE-1)) {
    if (ar[i] <= ar[j]) {
      res[k] = ar[i];
      i++;
    } else {
      res[k] = ar[j];
      j++;
    }
    k++;
  }

  while ((i<=SIZE/2-1)) {
    res[k] = ar[i];
    i++;
    k++;
  }
  while ((j<=SIZE-1)) {
    res[k] = ar[j];
    j++;
    k++;
  }

  //for(i=0; i<=SIZE-1; i++) {
    //printf("%d\n", res[i]);
  //}
  free(ar);
  return 0;
}
```

Compile the program

```
$  gcc -o thr_sort -lpthread thr_sort.c
```

Run the program & time it.

```
$ time ./thr_sort

real    0m10.564s
user    0m21.115s
sys     0m0.003s
```

You can notice that there is a lot of benefit by multi-threading the selection sort.

Module 5 : Pthreads Synchronization Primitives

Mutual Exclusion

Prototype:

```
#include <pthread.h>

int  pthread_mutex_init(pthread_mutex_t   *restrict   mutex,
                                const pthread_mutexattr_t *restrict attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
pthread_mutex_t mutex= PTHREAD_MUTEX_INITIALIZER;
```

Parameters:

Parameter	Direction	Description
mutex	out	pointer to mutex variable which is of type pthread_mutex_t
attr	in	mutex attributes. May be NULL, in which case, default attributes are used.

Return Value:
If successful, the *pthread_mutex_init()* and *pthred_mutex_destroy()* functions return 0. Otherwise, an error number is returned to indicate the error.

Description:
The pthread_mutex_init() function initializes the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

The pthread_mutex_destroy() function destroys the mutex object referenced by *mutex*; the mutex object becomes, in effect, uninitialized. A destroyed mutex object can be re-initialized using pthread_mutex_init().

Definition cum initialization of mutex can be done using PTHREAD_MUTEX_INITIALIZER macro.

Prototype:

#include <pthread.h>

*int pthread_mutex_lock(pthread_mutex_t *mutex);*

*int pthread_mutex_trylock(pthread_mutex_t *mutex);*

*int pthread_mutex_unlock(pthread_mutex_t *mutex);*

Parameters:

Parameter	Direction	Description
mutex	in	pointer to an initialized mutex. Must not be NULL.

Return Value:
If successful, the *pthread_mutex_lock() and pthread_mutex_unlock()* functions returns 0. Otherwise, an error number is returned to indicate the error.

The *pthread_mutex_trylock()* function returns 0 if a lock on the mutex object referenced by mutex is acquired. Otherwise, an error number is returned to indicate the error.

Description:
The mutex object referenced by mutex is locked by calling pthread_mutex_lock(). If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

The pthread_mutex_trylock() function is identical to pthread_mutex_lock() except that if the mutex object referenced by mutex is currently locked (by any thread, including the current thread), the call returns immediately.

The pthread_mutex_unlock() function releases the mutex object referenced by mutex. If there are threads blocked on the mutex object referenced by mutex when pthread_mutex_unlock() is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread will acquire the mutex.

Example : Salary and Expense Program using mutex.

.

```
$ vi sal_exp.c

#include <stdio.h>
#include <unistd.h>
#include <math.h>
#include <pthread.h>

#define SALARY 10000
unsigned int curr_bal=0;
int shutdown=0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void*
deposits(void *arg)
{
  do {
    pthread_mutex_lock(&mutex);
    curr_bal += SALARY;
    pthread_mutex_unlock(&mutex);
    sleep(30);
  } while(1);
}

unsigned int
withdraw(unsigned int amount)
{
  unsigned int withdrawn = 0;

  pthread_mutex_lock(&mutex);
  if (amount<=curr_bal) {
    curr_bal -= amount;
    withdrawn += amount;
  }
  pthread_mutex_unlock(&mutex);

  return withdrawn;
}

unsigned int
get_curr_bal()
{
  unsigned int retval = 0;

  pthread_mutex_lock(&mutex);
  retval = curr_bal;
  pthread_mutex_unlock(&mutex);

  return retval;
}

int main(void)
{
  pthread_t  th1;
  unsigned int withdrawn = 0, amount;

  (void) pthread_create(&th1, NULL, deposits, NULL);
  sleep(1);
  do {
    printf("\nCurrent Balance: %d, Enter amount to be withdrawn: ",
           get_curr_bal());
    scanf("%d", &amount);
    withdrawn = withdraw(amount);
    printf("Withdrawn amount=%d\n", withdrawn);
  } while(1);

  return 0;
}
```

Compile the program

```
$ gcc -o sal_exp –lpthread sal_exp.c
```

Run the program

```
$./sal_exp

Current Balance: 10000, Enter amount to be withdrawn: 1000
Withdrawn amount=1000

Current Balance: 9000, Enter amount to be withdrawn: 2000
Withdrawn amount=2000
```

Let the program run for 30 more seconds and enter 1000

```
Current Balance: 7000, Enter amount to be withdrawn: 1000
Withdrawn amount=1000

Current Balance: 16000, Enter amount to be withdrawn:
```

Notice that after 30 seconds 10,000 more got added to curr_bal.

Press ctrl+c to come out of the program.

Read/Write Synchronization

Prototype:
#include <pthread.h>

```
*#include <pthread.h>*

*int pthread_rwlock_init(pthread_rwlock_t  *restrict  rwlock,*
*const pthread_rwlockattr_t *restrict attr);*

*int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);*
*pthread_rwlock_t rwlock=PTHREAD_RWLOCK_INITIALIZER;*
```

Parameters:

Parameter	Direction	Description
rwlock	in	pointer to rwlock variable of type pthread_rwlock_t. Must not be NULL.
attr	in	rwlock attributes. May be NULL, in which case, default attributes are used.

Return Value:
If successful, the *pthread_rwlock_init()& pthred_rwlock_destroy()* functions return 0. Otherwise, an error number is returned to indicate the error.

Description:
The pthread_rwlock_init() function initializes the read-write lock referenced by rwlock with the attributes referenced by attr. If attr is NULL, the default read-write lock attributes are used; the effect is the same as passing the address of a default read-write lock attributes object. Once initialized, the lock can be used any number of times without being re-initialized. Upon successful initialization, the state of the read-write lock becomes initialized and unlocked.

The pthread_rwlock_destroy() function destroys the read-write lock object referenced by rwlock and releases any resources used by the lock.

Definition cum initialization of rwlock can be done using PTHREAD_RWLOCK_INITIALIZER macro.

Prototype:

#include <pthread.h>

*int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);*
*int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);*

*int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);*
*int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);*

Parameters:

Parameter	Direction	Description
rwlock	In	pointer to a initialized rwlock. Must not be NULL.

Return Value:

If successful, the *pthread_rwlock_rdlock()* and *pthread_rwlock_wrlock()* functions return 0. Otherwise, an error number is returned to indicate the error.

The *pthread_rwlock_tryrdlock()* and *pthread_rwlock_trywrlock()* functions return 0 if the lock for reading / writing respectively on the read-write lock object referenced by rwlock is acquired. Otherwise an error number is returned to indicate the error.

Description:

The pthread_rwlock_rdlock() function applies a read lock to the read-write lock referenced by rwlock. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock.

The calling thread does not acquire the lock if a writer holds the lock or if writers of higher or equal priority are blocked on the lock; otherwise, the calling thread acquires the lock. If the read lock is not acquired, the calling thread blocks until it can acquire the lock.

The pthread_rwlock_tryrdlock() function applies a read lock like the pthread_rwlock_rdlock() function, with the exception that the function fails if the equivalent pthread_rwlock_rdlock() call would have blocked the calling thread. In no case will the pthread_rwlock_tryrdlock() function ever blocks. It always either acquires the lock or fails and returns immediately.

The pthread_rwlock_wrlock() function applies a write lock to the read-write lock referenced by rwlock. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock rwlock. Otherwise, the thread blocks until it can acquire the lock.

The pthread_rwlock_trywrlock() function applies a write lock like the pthread_rwlock_wrlock() function, with the exception that the function fails if any thread currently holds rwlock (for reading or writing).

Writers are favored over readers of the same priority to avoid writer starvation.

Prototype:

#include <pthread.h>

*int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);*

Parameters:

Parameter	Direction	Description
rwlock	In	pointer to a initialized & locked rwlock. Must not be NULL

Return Value:

If successful, the *pthread_rwlock_unlock*() function returns 0. Otherwise, an error number is returned to indicate the error.

The pthread_rwlock_unlock() function is called to release a lock held on the read-write lock object referenced by rwlock.

If this function is called to release a read lock from the read-write lock object and there are other read locks currently held on this read-write lock object, the read-write lock object remains in the read locked state. If this function releases the calling thread's last read lock on this read-write lock object, then the calling thread is no longer one of the owners of the object. If this function releases the last read lock for this read-write lock object, the read-write lock object will be put in the unlocked state with no owners.

If this function is called to release a write lock for this read-write lock object, the read-write lock object will be put in the unlocked state with no owners.

If the call to the pthread_rwlock_unlock() function results in the read-write lock object becoming unlocked and there are multiple threads waiting to acquire the read-write lock object for writing, the scheduling policy is used to determine which thread acquires the read-write lock object for writing. If there are multiple threads waiting to acquire the read-write lock object for reading, the scheduling policy is used to determine the order in which the waiting threads acquire the read-write lock object for reading. If there are multiple threads blocked on rwlock for both read locks and write locks, it is unspecified whether the readers acquire the lock first or whether a writer acquires the lock first.

Example: Salary and Expense program using read-write lock.

```
$ vi sal_exp_rwlock.c

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

#define SALARY 10000
unsigned int curr_bal=0;

pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;

void*
deposits(void *arg)
{
  do {
    pthread_rwlock_wrlock(&rwlock);
    curr_bal += SALARY;
    pthread_rwlock_unlock(&rwlock);
    sleep(30);
  } while(1);
}

int
withdraw(unsigned int amount)
{
  unsigned int withdrawn = 0;

  pthread_rwlock_wrlock(&rwlock);
  if (amount<=curr_bal) {
    curr_bal -= amount;
    withdrawn += amount;
  }
  pthread_rwlock_unlock(&rwlock);

  return withdrawn;
}

unsigned int
get_curr_bal()
{
  unsigned int retval = 0;

  pthread_rwlock_rdlock(&rwlock);
  retval = curr_bal;
  pthread_rwlock_unlock(&rwlock);

  return retval;
}

int main(void)
{
  pthread_t  th1;
  unsigned int withdrawn = 0, amount;

  (void) pthread_create(&th1, NULL, deposits, NULL);
  sleep(1);
  do {
    printf("\nCurrent Balance: %u, Enter amount to be withdrawn: ",
           get_curr_bal());
    scanf("%u", &amount);
    withdrawn = withdraw(amount);
    printf("Withdrawn amount=%u\n", withdrawn);
  } while(1);

  return 0;
}
```

Compile the program

```
$ gcc -o sal_exp_rwlock –lpthread sal_exp_rwlock.c
```

Run the program

```
$./sal_exp_rwlock

Current Balance: 10000, Enter amount to be withdrawn: 1000
Withdrawn amount=1000

Current Balance: 9000, Enter amount to be withdrawn: 2000
Withdrawn amount=2000
```

Let the program run for 30 more seconds and enter 1000

```
Current Balance: 7000, Enter amount to be withdrawn: 1000
Withdrawn amount=1000

Current Balance: 16000, Enter amount to be withdrawn:
```

Notice that after 30 seconds 10,000 more got added to curr_bal.

Press ctrl+c to come out of the program.

Conditional Variables

Prototype:

#include <pthread.h>
*int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);*

*int pthread_cond_destroy(pthread_cond_t *cond);*
pthread_cond_t cond= PTHREAD_COND_INITIALIZER;

Parameters:

Parameter	Direction	Description
cond	out	pointer to condvar variable of type pthread_cond_t. Must not be NULL.
attr	in	condvar attributes. May be NULL, in which case, default attributes are used.

Return Value:
If successful, the *pthread_cond_init(), pthread_cond_destroy()* functions return 0. Otherwise, an error number is returned to indicate the error.

Description:
The function pthread_cond_init() initializes the condition variable referenced by cond with attributes referenced by attr. If attr is NULL, the default condition variable attributes are used; the effect is the same as passing the address of a default condition variable attributes object.

The function pthread_cond_destroy() destroys the given condition variable specified by cond; the object becomes, in effect, uninitialized. A destroyed condition variable object can be re-initialized using pthread_cond_init()

Prototype:
#include <pthread.h>

```
    #include <pthread.h>

    int pthread_cond_wait(pthread_cond_t *restrict cond,
        pthread_mutex_t *restrict mutex);

    int pthread_cond_timedwait(pthread_cond_t *restrict cond,
        pthread_mutex_t *restrict mutex,
        const struct timespec *restrict abstime);
```

Parameters:

Parameter	Direction	Description
cond	in	pointer to a initialized condvar. Must not be NULL
mutex	in	pointer to a initialized mutex. Must not be NULL
abstime	in	.absolute time for timeout

Return Value:
If successful, the *pthread_cond_wait() and pthread_cond_timedwait()* functions return 0. Otherwise, an error number is returned to indicate the error.

Description:
The pthread_cond_wait() and pthread_cond_timedwait() functions are used to block on a condition variable. They are called with mutex locked by the calling thread or undefined behavior will result.

These functions atomically release mutex and cause the calling thread to block on the condition variable cond. Atomically here means ``atomically with respect to access by another thread to the mutex and then the condition variable." That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to pthread_cond_signal() or pthread_cond_broadcast() in that thread behaves as if it were issued after the about-to-block thread has blocked.

When using condition variables there is always a boolean predicate, an invariant, associated with each condition wait that must be true before the thread should proceed. Spurious wakeups from the pthread_cond_wait() or pthread_cond_timedwait() functions could occur. Since the return from pthread_cond_wait() or pthread_cond_timedwait() does not imply anything about the value of this predicate, the predicate should always be reevaluated.

The order in which blocked threads are awakened by pthread_cond_signal() or pthread_cond_broadcast() is determined by the scheduling policy.

The pthread_cond_timedwait() function is the same as pthread_cond_wait() except that an error is returned if the absolute time specified by abstime passes (that is, system time equals or exceeds abstime) before the condition cond is signaled or broadcast, or if the absolute time specified by abstime has already been passed at the time of the call. The abstime argument is of type struct timespec, defined in time.h(3HEAD). When such time-outs occur, pthread_cond_timedwait() will nonetheless release and reacquire the mutex referenced by mutex.

Prototype:
#include <pthread.h>

#include <pthread.h>

*int pthread_cond_signal(pthread_cond_t *cond);*

*int pthread_cond_broadcast(pthread_cond_t *cond);*

Parameters:

Parameter	Direction	Description
cond	in	pointer to a initialized condvar. Must not be NULL

Return Value:
If successful, the *pthread_cond_signal(),pthread_cond_broadcastl()* functions return 0. Otherwise, an error number is returned to indicate the error.

Description:
These two functions are used to unblock threads blocked on a condition variable.

The pthread_cond_signal() call unblocks at least one of the threads that are blocked on the specified condition variable cond (if any threads are blocked on cond).

The pthread_cond_broadcast() call unblocks all threads currently blocked on the specified condition variable cond.

If more than one thread is blocked on a condition variable, the scheduling policy determines the order in which threads are unblocked. When each thread unblocked as a result of a pthread_cond_signal() or pthread_cond_broadcast() returns from its call to pthread_cond_wait() or pthread_cond_timedwait(), the thread owns the mutex with which it called pthread_cond_wait() or pthread_cond_timedwait(). The thread(s) that are unblocked contend for the mutex according to the scheduling policy (if applicable), and as if each had called pthread_mutex_lock().

The pthread_cond_signal() or pthread_cond_broadcast() functions may be called by a thread whether or not it currently owns the mutex that threads calling pthread_cond_wait() or pthread_cond_timedwait() have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex is locked by the thread calling pthread_cond_signal() or pthread_cond_broadcast().

The pthread_cond_signal() and pthread_cond_broadcast() functions have no effect if there are no threads currently blocked on cond.

Example : Client Server program. In this program Client submits requests and Server services requests asynchronously. The program also has shutdown facility to gracefully bring down the server.

```
$ vi client_server_cond.c

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


pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;

int shutdown=0;
unsigned int req_ctr = 0, pending_requests = 0;

void*
server(void *arg)
{
  char cmd[32];

  pthread_mutex_lock(&mutex);
  do {
    while (!shutdown && pending_requests == 0) {
      pthread_cond_wait(&cond, &mutex);
    }
    if (shutdown && pending_requests == 0) {
      break;
    }
    req_ctr++;
    sprintf(cmd, "touch serviced_request%d",req_ctr);
    system(cmd);
    sleep(1);
    pending_requests--;
  } while(1);
  pthread_mutex_unlock(&mutex);
}

void
submit_request(unsigned int num_req)
{
  pthread_mutex_lock(&mutex);
  pending_requests += num_req;
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mutex);
}

void
shutdown_server()
{
  pthread_mutex_lock(&mutex);
  shutdown = 1;
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mutex);
}
```

```
int main(void)
{
  pthread_t  th1;
  unsigned num_req;

  (void) pthread_create(&th1, NULL, server, NULL);

  do {
    printf("\nEnter no of requests for server [0-n]");
    scanf("%u", &num_req);
    if (num_req==0) {
      break;
    }
    submit_request(num_req);
  } while(1);

  shutdown_server();

  (void) pthread_join(th1, NULL);

  return 0;
}
```

Compile the program

```
$ gcc –o client_server_cond -lpthread -g client_server_cond.c
```

Run the program

```
$ ./client_server_cond

Enter no of requests for server [0-n]2

Enter no of requests for server [0-n]3

Enter no of requests for server [0-n]4

Enter no of requests for server [0-n]0
```

See the result of server’s request servicing.

```
$ ls -l
…
…
-rw-r--r--. 1 maruthisi fam    0 Jun 18 08:03 serviced_request1
-rw-r--r--. 1 maruthisi fam    0 Jun 18 08:03 serviced_request2
-rw-r--r--. 1 maruthisi fam    0 Jun 18 08:03 serviced_request3
-rw-r--r--. 1 maruthisi fam    0 Jun 18 08:03 serviced_request4
-rw-r--r--. 1 maruthisi fam    0 Jun 18 08:03 serviced_request5
-rw-r--r--. 1 maruthisi fam    0 Jun 18 08:03 serviced_request6
-rw-r--r--. 1 maruthisi fam    0 Jun 18 08:03 serviced_request7
-rw-r--r--. 1 maruthisi fam    0 Jun 18 08:03 serviced_request8
-rw-r--r--. 1 maruthisi fam    0 Jun 18 08:03 serviced_request9
…
…
```

What’s Next?

The content & examples given in this material covers fundamentals of multi-threaded programming & pthreads library. It also covered synchronization primitives in pthread library with examples. There are other synchronization primitives not covered viz., spin locks (see Appendix). Multi-process programming is another paradigm to exploit parallelism of SMP machines. The workshop doesnt cover the Multi-process programming, because it is already covered as part of curriculum. Yet another paradigm is multi-process and muti-threaded programming.

Hands-on Lab

Problem: Modify the client server program, to make the client be able to submit request while server is servicing a request. In other words, system() function and the sleep should not be with the lock held.

Problem: Modify the above client server program, to have multiple clients submitting requests and one server servicing requests.

Problem: Modify the above client server program, to have multiple clients submitting requests and multiple servers servicing requests.

Module 6 : Thread Design Patterns

Thread Design Patterns

Design Pattern:
Common ways of structuring programs

Thread Design Patterns:
Common ways of structuring programs using threads

Some Patterns :

- Boss/workers model
- Pipeline model

Design Pattern:

Common ways of structuring programs

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" – Christopher Alexander [1977]

i.e., Patterns == {"general problem", solution} pairs in a context

Thread Design Patterns:

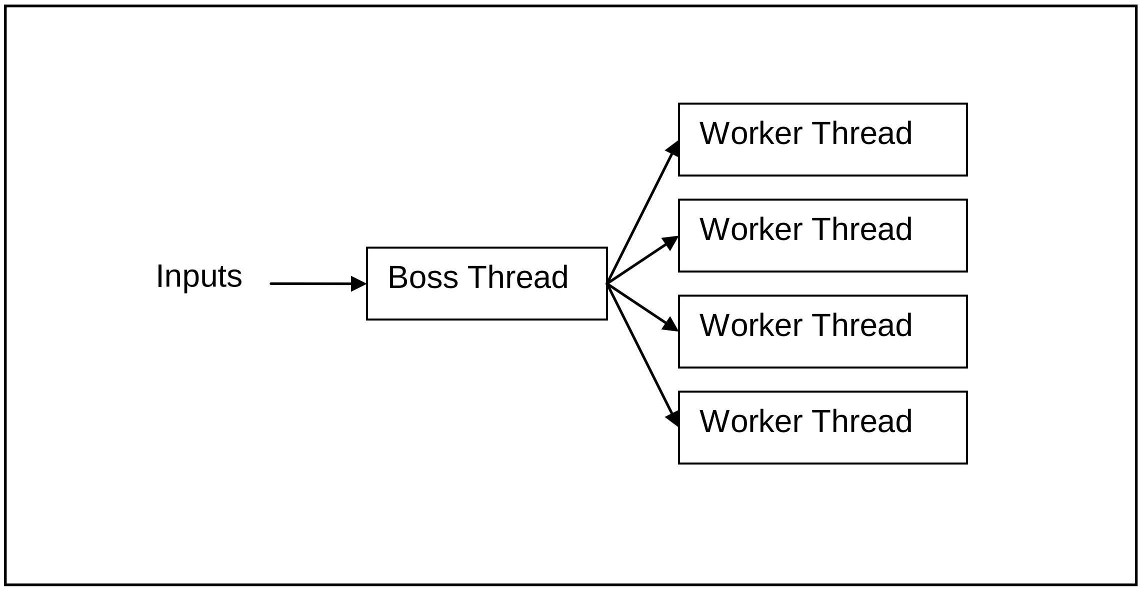
Common ways of structuring programs using threads

i.e., Thread Design Patterns == { "general design problem", solution} pairs in
threaded software design

A few common patterns have emerged that are useful in different threads applications

- Boss/Workers model
- Pipeline model

Boss/Workers Model

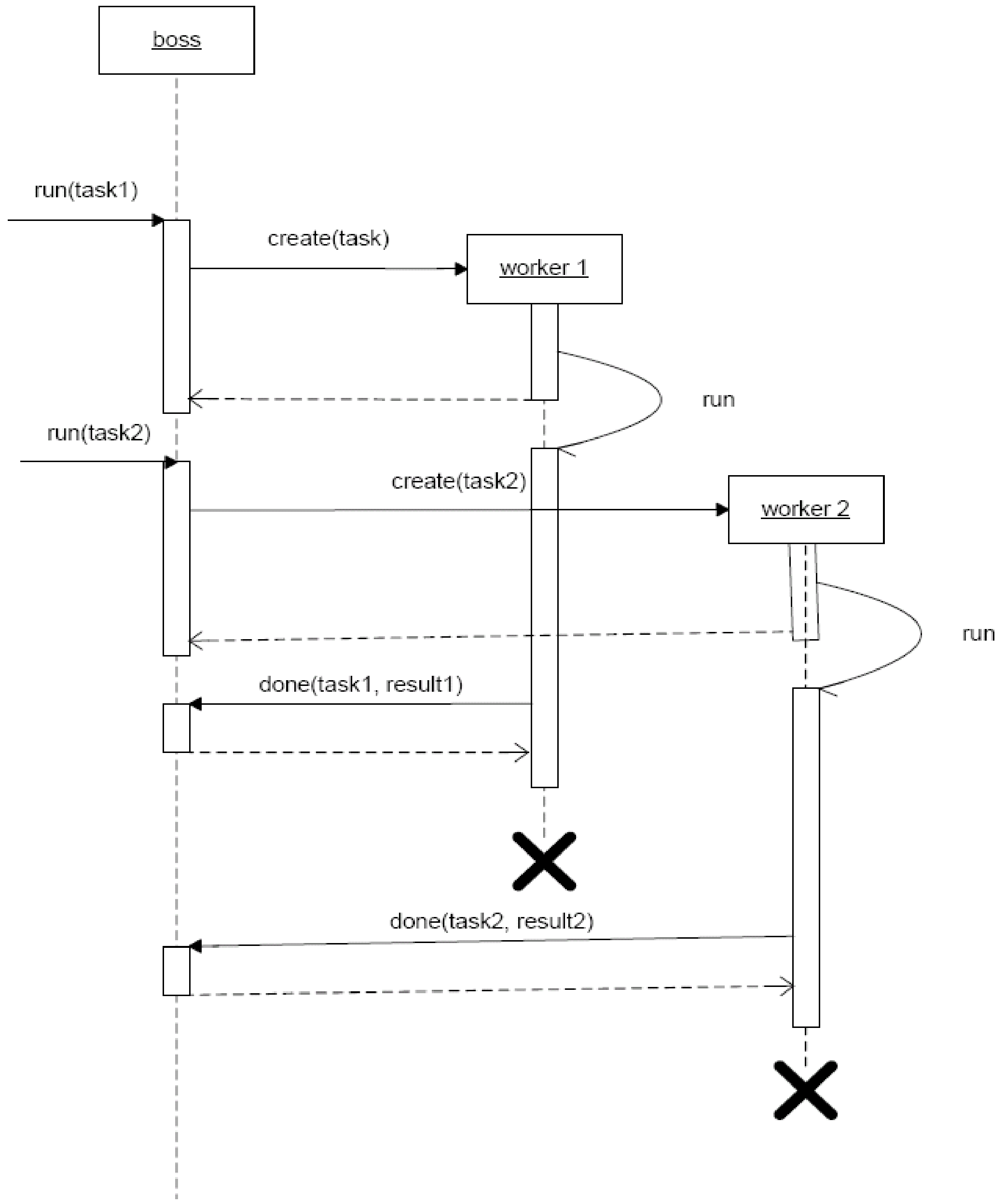


Boss/Worker Model: In this type of model, one thread functions as the boss and assigns tasks to the worker threads. The worker thread signals the boss on completion of its task. Alternatively, the boss polls the workers periodically and assigns tasks when a worker has completed its task.

Variants of Boss / Worker Model

- Boss/Worker I
- Boss/Worker II

Boss/Workers I



Boss / Worker I : all tasks come into a single "Boss Thread" who passes the tasks off to worker threads that it creates on the fly.

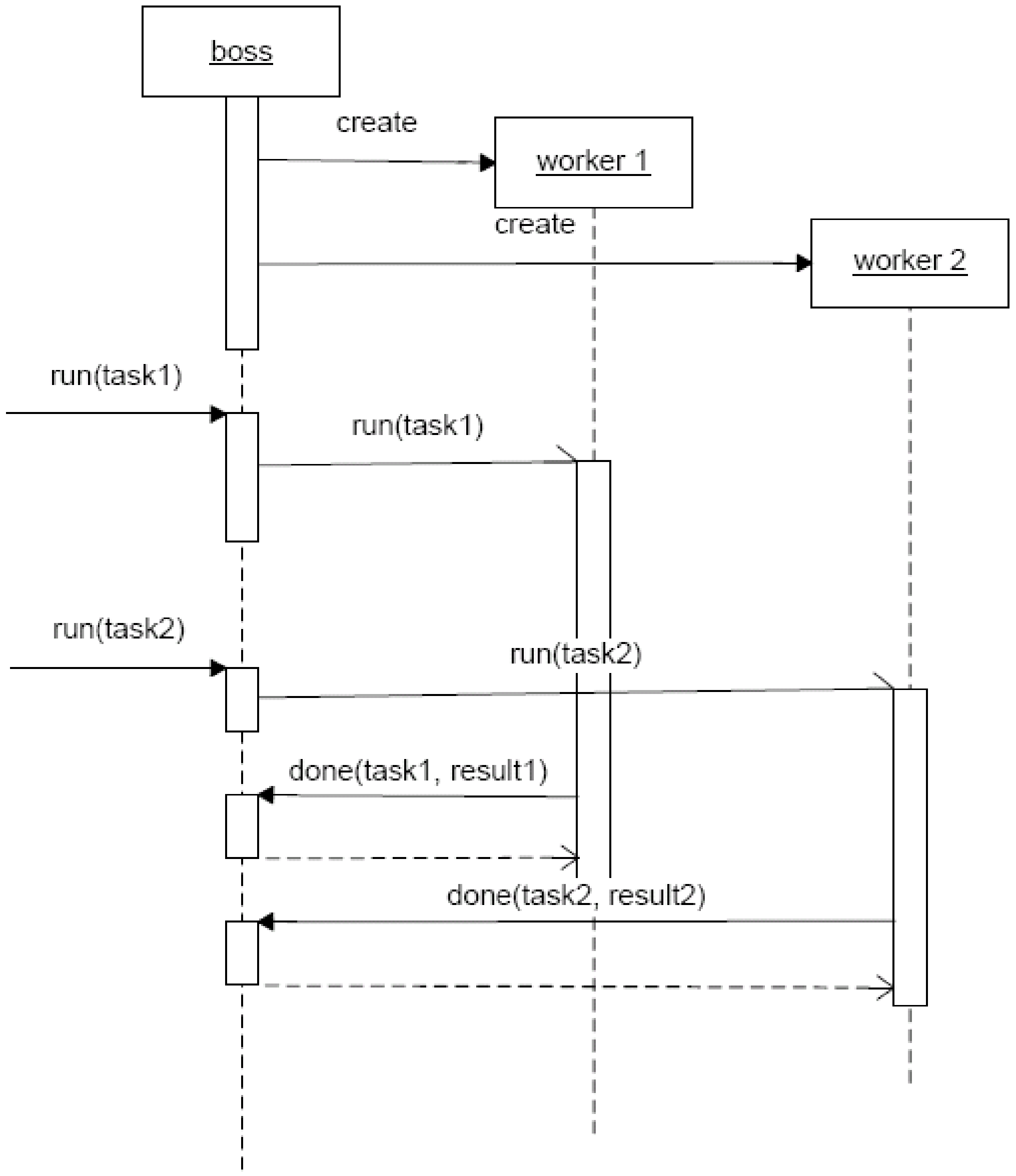
Advantage:

- simplicity

Disadvantages:

- no bound on number of workers (sidenote: Sun Pthreads implementation has a default 1MB stack per thread)
- Thread creation and destruction overhead
- potential for contention if requests have interdependencies

Boss/Workers II



Boss / Worker II : all tasks come into a single “Boss Thread” who passes the tasks off to worker threads from a “thread pool” that the Boss created up front. This model is also called Workpile or WorkQueue or ThreadPool

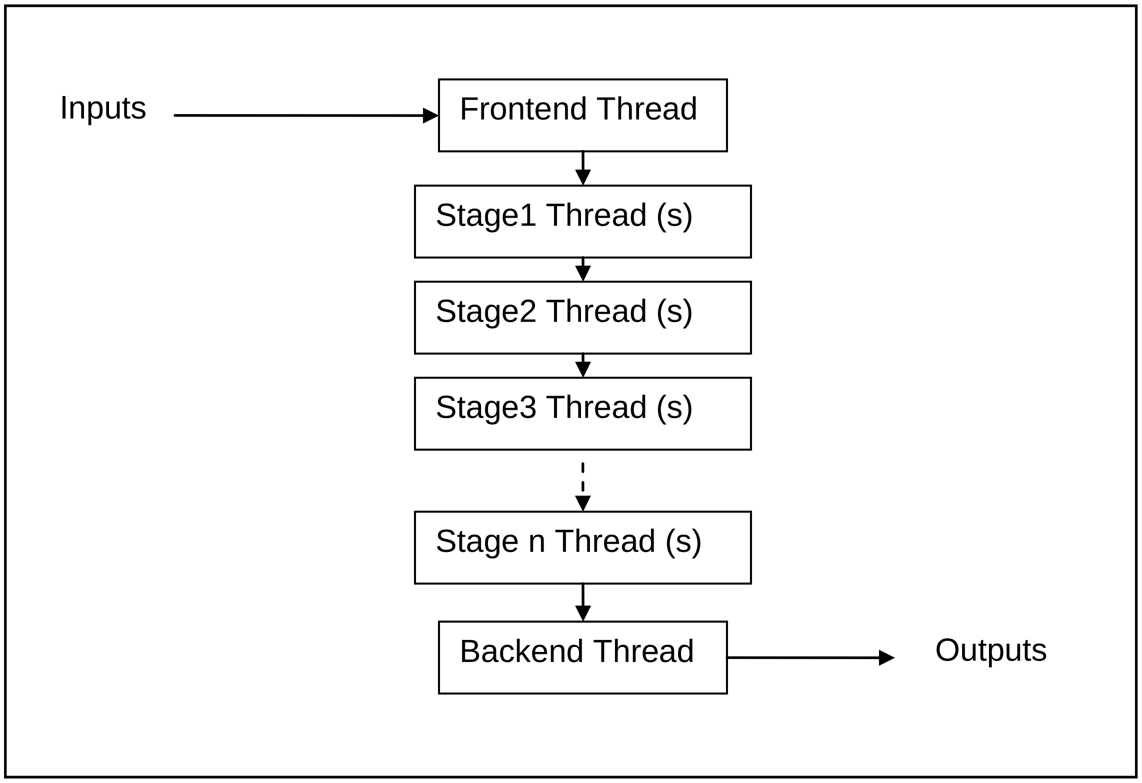
Advantages:

- all worker threads are created up front (a fixed number of them)
- Avoids overhead of thread creation and destruction

Disadvantages:

- a bit complex. (when all workers go to sleep, boss needs to wakeup when there is a task to be done)

Pipeline



Pipelining Model: In this model, a task is divided into steps. The steps must be performed in sequence. However, the program is designed to produce multiple instances of the output, and the steps are designed to operate in parallel. In other words "do some work, pass the partial result to the next thread". Just like an assembly line or a processor pipeline

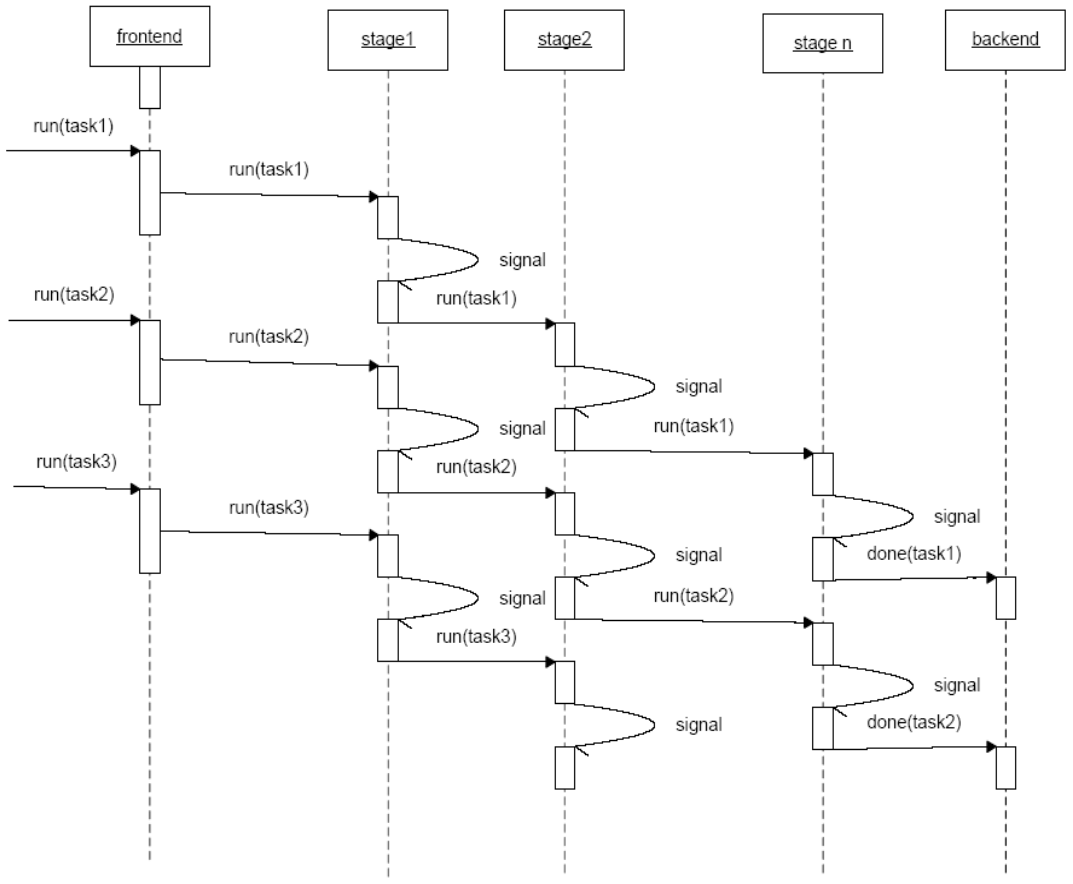
Advantages:

- simplicity (trivial synchronization)
- low overhead for thread creation

Disadvantages:

- hand tuning is needed for determining the right number of pipeline stages.
- hardwired limit in the degree of parallelization.
- overall throughput is limited by slowest stage.

Pipeline (contd…)



Other Patterns

A few other patterns that are used in concurrent programming are [Schmidt]

- Active Object.
- Monitor Object.
- Half-Sync Half-Async.
- Leaders / Followers
- Thread-Specific Storage

Appendix A : Posix pthreads

The following table list all POSIX pthreads functions.

Functions Related to Creation
pthread_create()
pthread_attr_init()
pthread_attr_setdetachstate()
pthread_attr_getdetachstate()
pthread_attr_setinheritsched()
pthread_attr_getinheritsched()
pthread_attr_setschedparam()
pthread_attr_getschedparam()
pthread_attr_setschedpolicy()
pthread_attr_getschedpolicy()
pthread_attr_setscope()
pthread_attr_getscope()
pthread_attr_setstackaddr()
pthread_attr_getstackaddr()
pthread_attr_setstacksize()
pthread_attr_getstacksize()
pthread_attr_getguardsize()
pthread_attr_setguardsize()
pthread_attr_destroy()

Functions Related to Exit
pthread_exit()
pthread_join()
pthread_detach()

Functions Related to Thread Specific Data
pthread_key_create()
pthread_setspecific()
pthread_getspecific()
pthread_key_delete()

Functions Related to Signals
pthread_sigmask()
pthread_kill()

Functions Related to IDs
pthread_self()
pthread_equal()

Functions Related to SpinLocks
pthread_spin_init()
pthread_spin_lock()
pthread_spin_trylock()
pthread_spin_unlock()
pthread_spin_destroy()

Functions Related to Scheduling
pthread_setconcurrency()
pthread_getconcurrency()
pthread_setschedparam()
pthread_setschedprio()
pthread_getschedparam()

Functions Related to Cancellation
pthread_cancel()
pthread_setcancelstate()
pthread_setcanceltype()
pthread_testcancel()
pthread_cleanup_pop()
pthread_cleanup_push()

Functions Related to Mutexes
pthread_mutex_init()
pthread_mutexattr_init()
pthread_mutexattr_setpshared()
pthread_mutexattr_getpshared()
pthread_mutexattr_setprotocol()
pthread_mutexattr_getprotocol()
pthread_mutexattr_setprioceiling()
pthread_mutexattr_getprioceiling()
pthread_mutexattr_settype()
pthread_mutexattr_gettype()

pthread_mutexattr_setrobust()
pthread_mutexattr_getrobust()
pthread_mutexattr_destroy()
pthread_mutex_setprioceiling()
pthread_mutex_getprioceiling()
pthread_mutex_lock()
pthread_mutex_trylock()
pthread_mutex_unlock()
pthread_mutex_destroy()

Functions Related to Condition Variables
pthread_cond_init()
pthread_condattr_init()
pthread_condattr_setpshared()
pthread_condattr_getpshared()
pthread_condattr_destroy()
pthread_cond_wait()
pthread_cond_timedwait()
pthread_cond_signal()
pthread_cond_broadcast()
pthread_cond_destroy()

Functions Related to Reader/Writer Locking
pthread_rwlock_init()
pthread_rwlock_rdlock()
pthread_rwlock_tryrdlock()
pthread_rwlock_wrlock()
pthread_rwlock_trywrlock()
pthread_rwlock_unlock()
pthread_rwlock_destroy()
pthread_rwlockattr_init()
pthread_rwlockattr_destroy()
pthread_rwlockattr_getpshared()
pthread_rwlockattr_setpshared()

Functions Related to Semaphores
sem_init()
sem_open()
sem_close()
sem_wait()

sem_trywait()
sem_post()
sem_getvalue()
sem_unlink()
sem_destroy()

Functions Related to fork( ) Clean Up
pthread_atfork()

Functions Related to Limits
pthread_once()

References

[SUN] Multithreaded Programming Guide, sep 2008 – Sun Microsystems.

[Nichols] pThreads Programming by Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell - O'Reilly & Associates, Inc

[Stevens] Advanced Programming in the UNIX® Environment: Second Edition, by W. Richard Stevens, Stephen A. Rago - Addison Wesley Professional

[HUGHES] Parallel and Distributed Programming Using C++ by Cameron Hughes, Tracey Hughes - Addison Wesley Longman

[Gray] Interprocess Communications in Linux : The nooks & Crannies by John Shapley Gray - PH PTR

[Jones] GNU/Linux Application Programming, by M. Tim Jones - Charles River Media

[Schmidt] Pattern Oriented Software Architecture – volume 2: Patterns for concurrent and networked objects – John Wiley & Sons.