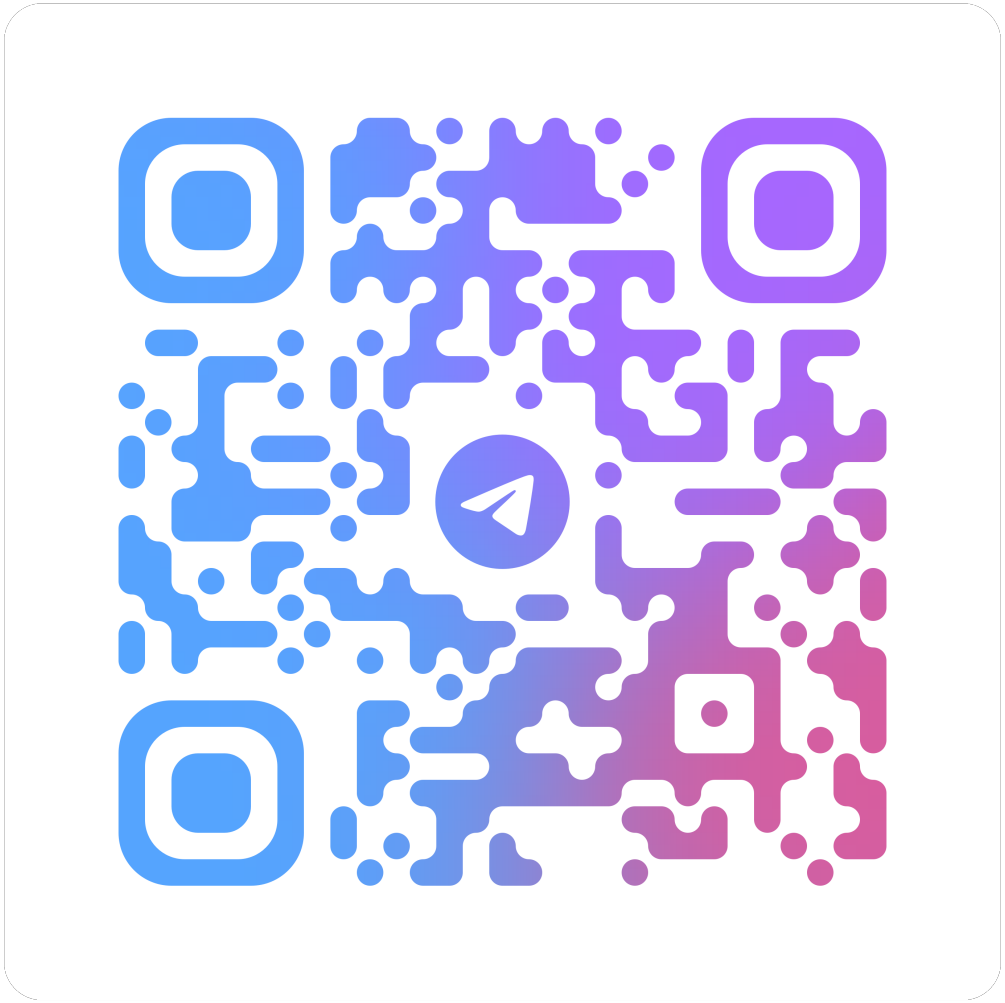
CSL 301
OPERATING SYSTEMS

Instructor
Dr. Dhiman Saha

Image Source: Google

Logistics

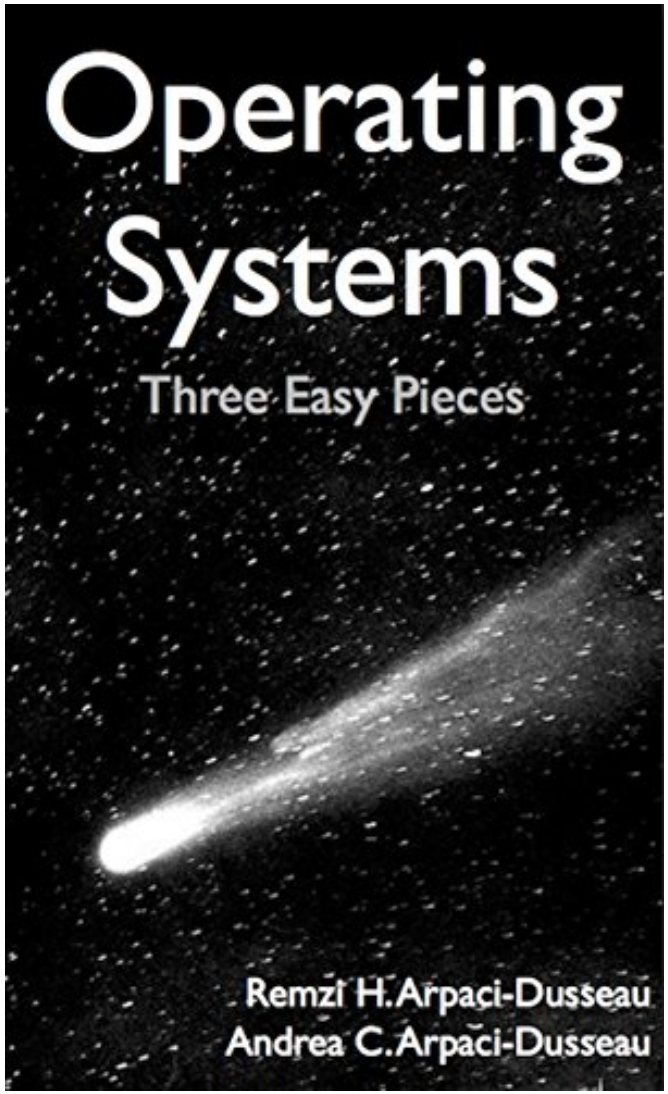
Telegram Group



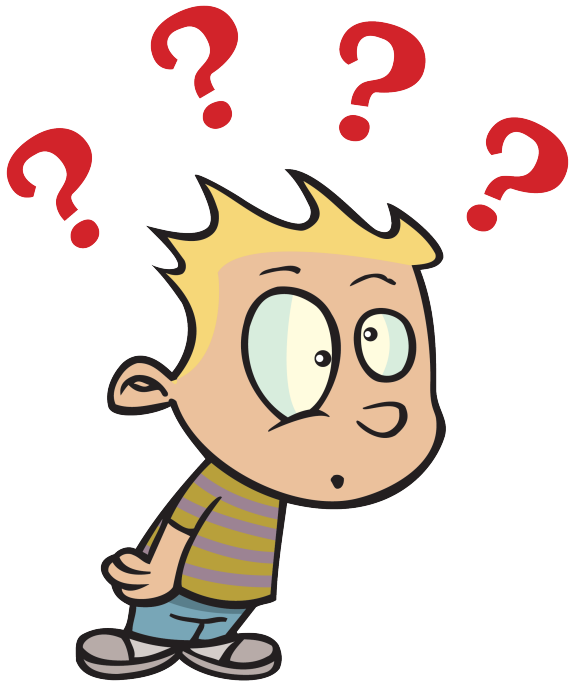
Course Format (LTP) : $3-0-2$			
Lecture	L102	Monday	9:30 AM
Lecture	L102	Tuesday	8:30 AM
Lab	ED1−320	Wednesday	3:30 PM
Lecture	L102	Thursday	9:30 AM

Note

- One Test Every-week on Lab-day
- Labs to be finished in class



- 7 hard copies available at institute library
- Soft-copy (chapter-wise) available freely on book-site
- Unofficial consolidated version available on GitHub
- Personal Copy:
  `https://pothi.com/pothi/book/remzi-h-arpaci-dusseau-operating-`



- OS?
- Why is it needed?
- Why should I study it?
- Is it difficult?
- Will I pass?

Image Credits: ComputerHope.com

- Middleware between user programs and system hardware

OS-CR Analogy Instructor $\leftrightarrow$ CR $\leftrightarrow$ Class

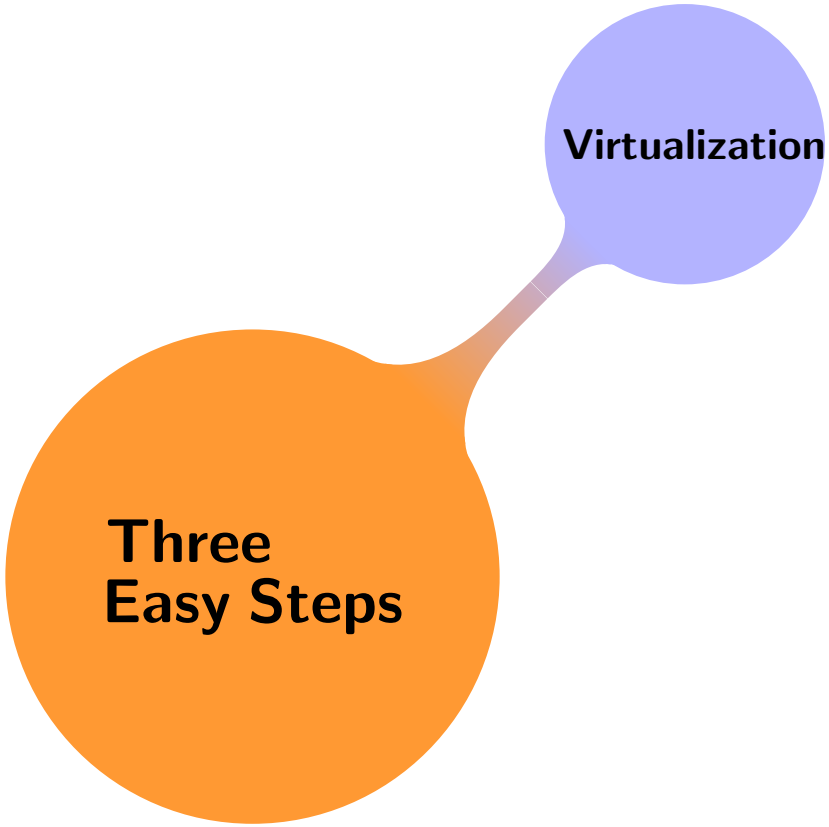
OS is a body of software To do what?

- To run programs easily
- Allowing programs to share memory
- Enabling programs to interact with devices
- So on and so forth.

- Also known as **supervisor** or the **master control program**

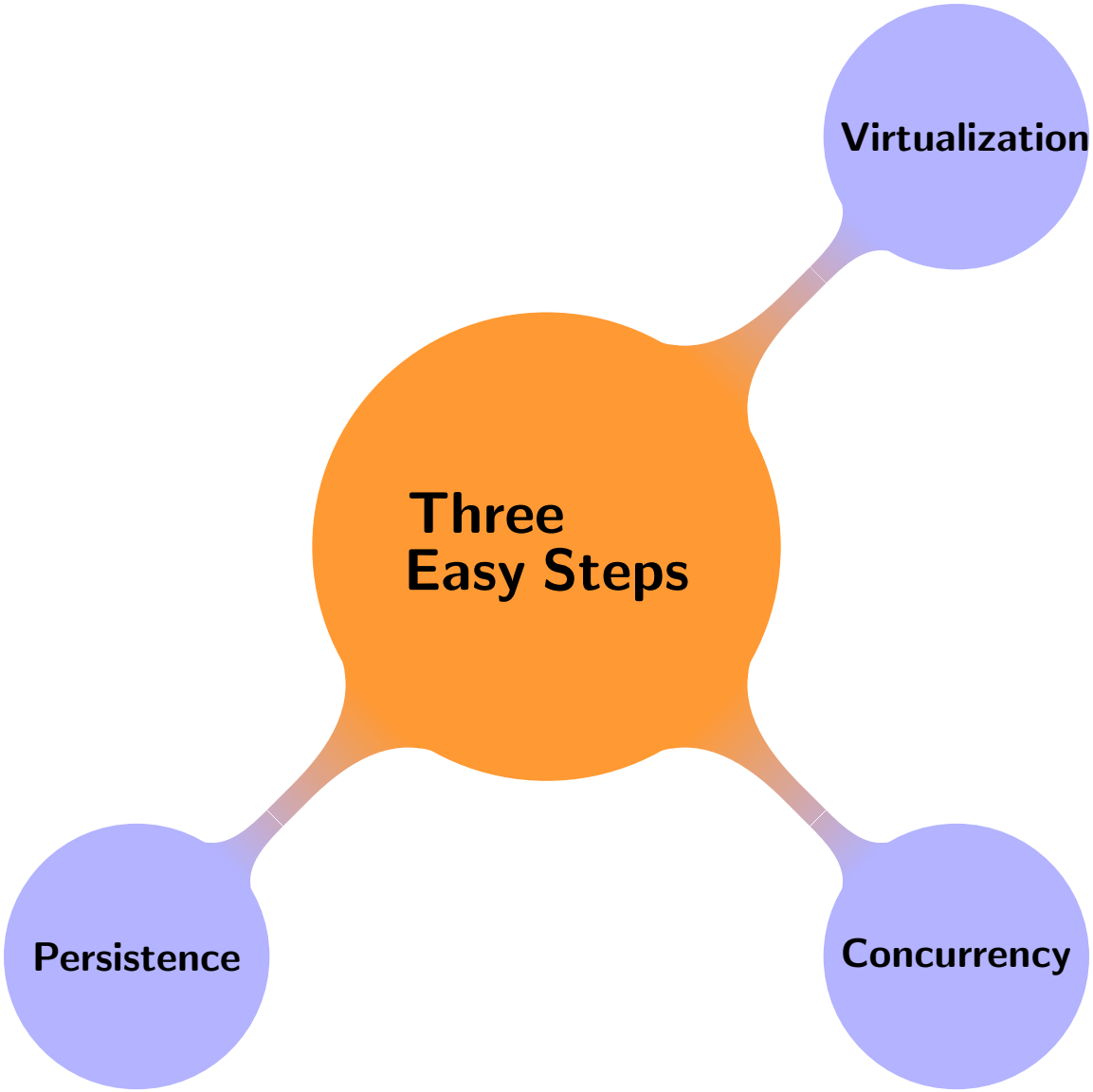
How does the OS do what is does?

Course Plan


Virtualization
Three
Easy Steps


Virtualization
Three
Easy Steps
Concurrency

Course Plan


Virtualization
Three
Easy Steps
Persistence
Concurrency


Security
Virtualization
Four
Easy Steps
Persistence
Concurrency

Virtualization

The Notion of Virtualization

The OS takes a physical resource such as

- the processor or
- memory or
- a disk

and transforms it into a more general, powerful, and easy-to-use **virtual** *form of itself*

- Earns OS the alternative name as a **virtual machine**
- The OS provides a **standard library** to applications
- The OS acts as a **resource manager** fairly and efficiently managing resources like the CPU, memory or the disk.

What happens when a program runs?

The Von Neumann model of computing

- It executes instructions

What happens when a program runs?

The Von Neumann model of computing

- It executes instructions
- The processor fetches an instruction from memory

What happens when a program runs?

The Von Neumann model of computing

- It executes instructions
- The processor fetches an instruction from memory
- Decodes it and

What happens when a program runs?

The Von Neumann model of computing

- It executes instructions
- The processor fetches an instruction from memory
- Decodes it and
- Executes it

What happens when a program runs?

The Von Neumann model of computing

- It executes instructions
- The processor fetches an instruction from memory
- Decodes it and
- Executes it
- Then processor moves to the next instruction
- Repeats this cycle until the program completes

What happens when a program runs?

The Von Neumann model of computing

- It executes instructions
- The processor fetches an instruction from memory
- Decodes it and
- Executes it
- Then processor moves to the next instruction
- Repeats this cycle until the program completes

Is this really that simple? Of course not!

- Lot of wild things will happen
- Primary motivation: ease of use

How to virtulalize resources?

How does the operating system virtualize resources?

- What mechanisms and policies are implemented by the OS to attain virtualization?
- How does the OS do so efficiently?
- What hardware support is needed?

CS250 will answer these questions!

Virtualizing The CPU

Simple Example — Code That Loops And Prints

```c
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "usage: cpu <string>\n");
                exit(1);
        }
        char *str = argv[1];

        while (1) {
                printf("%s\n", str);
                Spin(1);
        }
        return 0;
}
```

Simple Example — Code That Loops And Prints

```
1  #include <stdio.h>      Includes the standard I/O library for functions like 'printf'.
2  #include <stdlib.h>
3  #include "common.h"
4
5  int main(int argc, char *argv[])
6  {
7          if (argc != 2) {
8                  fprintf(stderr, "usage: cpu <string>\n");
9                  exit(1);
10         }
11         char *str = argv[1];
12
13         while (1) {
14                 printf("%s\n", str);
15                 Spin(1);
16         }
17         return 0;
18 }
```

Simple Example — Code That Loops And Prints

```
1  #include <stdio.h>
2  #include <stdlib.h>   Includes the standard library for functions like 'exit'.
3  #include "common.h"
4
5  int main(int argc, char *argv[])
6  {
7          if (argc != 2) {
8                  fprintf(stderr, "usage: cpu <string>\n");
9                  exit(1);
10         }
11         char *str = argv[1];
12
13         while (1) {
14                 printf("%s\n", str);
15                 Spin(1);
16         }
17         return 0;
18 }
```

Simple Example — Code That Loops And Prints

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4
5  int main(int argc, char *argv[])
6  {
7          if (argc != 2) {
8                  fprintf(stderr, "usage: cpu <string>\n");
9                  exit(1);
10         }
11         char *str = argv[1];
12
13         while (1) {
14                 printf("%s\n", str);
15                 Spin(1);
16         }
17         return 0;
18 }
```

Includes a local header file, likely containing the 'Spin()' function definition.

Simple Example — Code That Loops And Prints

```c
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "usage: cpu <string>\n");
                exit(1);
        }
        char *str = argv[1];

        while (1) {
                printf("%s\n", str);
                Spin(1);
        }
        return 0;
}
```

The program's entry point.
'argc' holds the count of command-line arguments
'argv' is an array of those argument strings.

Simple Example — Code That Loops And Prints

```c
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "usage: cpu <string>\n");
                exit(1);
        }
        char *str = argv[1];

        while (1) {
                printf("%s\n", str);
                Spin(1);
        }
        return 0;
}
```

Checks if #arguments is not equal to 2 (program name + one argument).

Simple Example — Code That Loops And Prints

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4
5  int main(int argc, char *argv[])
6  {
7          if (argc != 2) {
8                  fprintf(stderr, "usage: cpu <string>\n");
9                  exit(1);
10         }
11         char *str = argv[1];   Stores the first command-line argument in the 'str' variable.
12
13         while (1) {
14                 printf("%s\n", str);
15                 Spin(1);
16         }
17         return 0;
18 }
```

Simple Example — Code That Loops And Prints

```c
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "usage: cpu <string>\n");
                exit(1);
        }
        char *str = argv[1];

        while (1) {
                printf("%s\n", str);
                Spin(1);
        }
        return 0;
}
```

Starts an infinite loop, causing the program to run continuously.

Simple Example — Code That Loops And Prints

```c
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "usage: cpu <string>\n");
                exit(1);
        }
        char *str = argv[1];

        while (1) {
                printf("%s\n", str);
                Spin(1);
        }
        return 0;
}
```

The 'Spin(1)' function creates a busy-wait loop for 1 second, consuming CPU cycles without doing productive work.

Running a single program

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

Running Many Programs At Once

```
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...
```

Turning a single CPU (or small set of them) into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once is what we call **virtualizing** the CPU

Virtualizing The Memory

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5  int main(int argc, char *argv[]) {
6          if (argc != 2) {
7                  fprintf(stderr, "usage: mem <value>\n");
8                  exit(1);
9          }
10         int *p;
11         p = malloc(sizeof(int));
12         assert(p != NULL);
13         printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);

14         *p = atoi(argv[1]);
15         while (1) {
16                 Spin(1);
17                 *p = *p + 1;
18                 printf("(%d) value of p: %d\n", getpid(),*p);
19         }
20         return 0;
21 }
```

```
1  #include <unistd.h>   Includes POSIX operating system API for 'getpid()'
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5  int main(int argc, char *argv[]) {
6          if (argc != 2) {
7                  fprintf(stderr, "usage: mem <value>\n");
8                  exit(1);
9          }
10         int *p;
11         p = malloc(sizeof(int));
12         assert(p != NULL);
13         printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);

14         *p = atoi(argv[1]);
15         while (1) {
16                 Spin(1);
17                 *p = *p + 1;
18                 printf("(%d) value of p: %d\n", getpid(),*p);
19         }
20         return 0;
21 }
```

```
1  #include <unistd.h>
2  #include <stdio.h>    Includes standard input/output library
3  #include <stdlib.h>
4  #include "common.h"
5  int main(int argc, char *argv[]) {
6          if (argc != 2) {
7                  fprintf(stderr, "usage: mem <value>\n");
8                  exit(1);
9          }
10         int *p;
11         p = malloc(sizeof(int));
12         assert(p != NULL);
13         printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);

14         *p = atoi(argv[1]);
15         while (1) {
16                 Spin(1);
17                 *p = *p + 1;
18                 printf("(%d) value of p: %d\n", getpid(),*p);
19         }
20         return 0;
21 }
```

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>     Includes standard library for ‘malloc‘, ‘atoi‘, ‘exit‘
4  #include "common.h"
5  int main(int argc, char *argv[]) {
6          if (argc != 2) {
7                  fprintf(stderr, "usage: mem <value>\n");
8                  exit(1);
9          }
10         int *p;
11         p = malloc(sizeof(int));
12         assert(p != NULL);
13         printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);

14         *p = atoi(argv[1]);
15         while (1) {
16                 Spin(1);
17                 *p = *p + 1;
18                 printf("(%d) value of p: %d\n", getpid(),*p);
19         }
20         return 0;
21 }
```

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"          Includes a local header file for 'Spin()'
5  int main(int argc, char *argv[]) {
6          if (argc != 2) {
7                  fprintf(stderr, "usage: mem <value>\n");
8                  exit(1);
9          }
10         int *p;
11         p = malloc(sizeof(int));
12         assert(p != NULL);
13         printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);

14         *p = atoi(argv[1]);
15         while (1) {
16                 Spin(1);
17                 *p = *p + 1;
18                 printf("(%d) value of p: %d\n", getpid(),*p);
19         }
20         return 0;
21 }
```

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5  int main(int argc, char *argv[]) {
6          if (argc != 2) {
7                  fprintf(stderr, "usage: mem <value>\n");
8                  exit(1);
9          }
10         int *p;
11         p = malloc(sizeof(int));
12         assert(p != NULL);
13         printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);
14         *p = atoi(argv[1]);
15         while (1) {
16                 Spin(1);
17                 *p = *p + 1;
18                 printf("(%d) value of p: %d\n", getpid(),*p);
19         }
20         return 0;
21 }
```

Main function, entry point of the program

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5  int main(int argc, char *argv[]) {
6          if (argc != 2) {      Checks for the correct number of arguments
7                  fprintf(stderr, "usage: mem <value>\n");
8                  exit(1);
9          }
10         int *p;
11         p = malloc(sizeof(int));
12         assert(p != NULL);
13         printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);

14         *p = atoi(argv[1]);
15         while (1) {
16                 Spin(1);
17                 *p = *p + 1;
18                 printf("(%d) value of p: %d\n", getpid(),*p);
19         }
20         return 0;
21 }
```

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5  int main(int argc, char *argv[]) {
6          if (argc != 2) {
7                  fprintf(stderr, "usage: mem <value>\n");
8                  exit(1);
9          }
10         int *p;    Declares a pointer to an integer
11         p = malloc(sizeof(int));
12         assert(p != NULL);
13         printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);
14         *p = atoi(argv[1]);
15         while (1) {
16                 Spin(1);
17                 *p = *p + 1;
18                 printf("(%d) value of p: %d\n", getpid(),*p);
19         }
20         return 0;
21 }
```

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5  int main(int argc, char *argv[]) {
6          if (argc != 2) {
7                  fprintf(stderr, "usage: mem <value>\n");
8                  exit(1);
9          }
10         int *p;
11         p = malloc(sizeof(int));
12         assert(p != NULL);
13         printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);
14         *p = atoi(argv[1]);
15         while (1) {
16                 Spin(1);
17                 *p = *p + 1;
18                 printf("(%d) value of p: %d\n", getpid(),*p);
19         }
20         return 0;
21 }
```

Allocates memory for one integer on the heap

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5  int main(int argc, char *argv[]) {
6          if (argc != 2) {
7                  fprintf(stderr, "usage: mem <value>\n");
8                  exit(1);
9          }
10         int *p;
11         p = malloc(sizeof(int));
12         assert(p != NULL);
13         printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);
14         *p = atoi(argv[1]);
15         while (1) {
16                 Spin(1);
17                 *p = *p + 1;
18                 printf("(%d) value of p: %d\n", getpid(),*p);
19         }
20         return 0;
21 }
```

Asserts that the memory allocation was successful

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5  int main(int argc, char *argv[]) {
6          if (argc != 2) {
7                  fprintf(stderr, "usage: mem <value>\n");
8                  exit(1);
9          }
10         int *p;
11         p = malloc(sizeof(int));
12         assert(p != NULL);
13         printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);
                   Prints the process ID and the memory address
14         *p = atoi(argv[1]);
15         while (1) {
16                 Spin(1);
17                 *p = *p + 1;
18                 printf("(%d) value of p: %d\n", getpid(),*p);
19         }
20         return 0;
21 }
```

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5  int main(int argc, char *argv[]) {
6          if (argc != 2) {
7                  fprintf(stderr, "usage: mem <value>\n");
8                  exit(1);
9          }
10         int *p;
11         p = malloc(sizeof(int));
12         assert(p != NULL);
13         printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);
14
15         *p = atoi(argv[1]);
16         while (1) {
17                 Spin(1);
18                 *p = *p + 1;
19                 printf("(%d) value of p: %d\n", getpid(),*p);
20         }
21         return 0;
   }
```

Converts the command-line argument to an integer and stores it in the allocated memory

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5  int main(int argc, char *argv[]) {
6          if (argc != 2) {
7                  fprintf(stderr, "usage: mem <value>\n");
8                  exit(1);
9          }
10         int *p;
11         p = malloc(sizeof(int));
12         assert(p != NULL);
13         printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);
14
14         *p = atoi(argv[1]);
15         while (1) {    Starts an infinite loop
16                 Spin(1);
17                 *p = *p + 1;
18                 printf("(%d) value of p: %d\n", getpid(),*p);
19         }
20         return 0;
21 }
```

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5  int main(int argc, char *argv[]) {
6          if (argc != 2) {
7                  fprintf(stderr, "usage: mem <value>\n");
8                  exit(1);
9          }
10         int *p;
11         p = malloc(sizeof(int));
12         assert(p != NULL);
13         printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);
14         *p = atoi(argv[1]);
15         while (1) {
16                 Spin(1);
17                 *p = *p + 1;     Increments the value stored in the allocated memory
18                 printf("(%d) value of p: %d\n", getpid(),*p);
19         }
20         return 0;
21 }
```

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5  int main(int argc, char *argv[]) {
6          if (argc != 2) {
7                  fprintf(stderr, "usage: mem <value>\n");
8                  exit(1);
9          }
10         int *p;
11         p = malloc(sizeof(int));
12         assert(p != NULL);
13         printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);
14         *p = atoi(argv[1]);
15         while (1) {
16                 Spin(1);
17                 *p = *p + 1;
18                 printf("(%d) value of p: %d\n", getpid(),*p);
19         }
20         return 0;
21 }
```

Prints the process ID and the new value

Running The Memory Program Once

```
prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

Running The Memory Program Multiple Times

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

Illusion Vs Reality

Illusion

As far as the running program is concerned, it has physical memory all to itself.

Reality

Physical memory is a shared resource, managed by the operating system.

OS is virtualizing memory

- Each process accesses its own **private virtual address space**, which the OS **somehow** maps onto the physical memory of the machine.
- A memory reference within one running program does not affect the address space of other processes (or the OS itself)

Somehow?

CS250 will answer these questions!

Concurrency

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  volatile int counter = 0;
7  int loops;
8
9  void *worker(void *arg) {
10         int i;
11         for (i = 0; i < loops; i++) {
12                 counter++;
13         }
14         return NULL;
15 }
16
17 int main(int argc, char *argv[]) {
18         if (argc != 2) {
19                 fprintf(stderr, "usage: threads <loops>\n");
20                 exit(1);
21         }
22         loops = atoi(argv[1]);
23         pthread_t p1, p2;
24         printf("Initial value : %d\n", counter);
25         Pthread_create(&p1, NULL, worker, NULL);
26         Pthread_create(&p2, NULL, worker, NULL);
27         Pthread_join(p1, NULL);
28         Pthread_join(p2, NULL);
29         printf("Final value   : %d\n", counter);
30         return 0;
31 }
32 prompt> ./threads 10000
```

Simple Example — Modifying Shared Variable

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  volatile int counter = 0;     Declares a volatile integer 'counter' initialized to 0.
7  int loops;
8
9  void *worker(void *arg) {
10         int i;
11         for (i = 0; i < loops; i++) {
12                 counter++;
13         }
14         return NULL;
15 }
16
17 int main(int argc, char *argv[]) {
18         if (argc != 2) {
19                 fprintf(stderr, "usage: threads <loops>\n");
20                 exit(1);
21         }
22         loops = atoi(argv[1]);
23         pthread_t p1, p2;
24         printf("Initial value : %d\n", counter);
25         Pthread_create(&p1, NULL, worker, NULL);
26         Pthread_create(&p2, NULL, worker, NULL);
27         Pthread_join(p1, NULL);
28         Pthread_join(p2, NULL);
29         printf("Final value   : %d\n", counter);
30         return 0;
31 }
32 prompt> ./threads 10000
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  volatile int counter = 0;
7  int loops;     Declares an integer 'loops'.
8
9  void *worker(void *arg) {
10         int i;
11         for (i = 0; i < loops; i++) {
12                 counter++;
13         }
14         return NULL;
15 }
16
17 int main(int argc, char *argv[]) {
18         if (argc != 2) {
19                 fprintf(stderr, "usage: threads <loops>\n");
20                 exit(1);
21         }
22         loops = atoi(argv[1]);
23         pthread_t p1, p2;
24         printf("Initial value : %d\n", counter);
25         Pthread_create(&p1, NULL, worker, NULL);
26         Pthread_create(&p2, NULL, worker, NULL);
27         Pthread_join(p1, NULL);
28         Pthread_join(p2, NULL);
29         printf("Final value   : %d\n", counter);
30         return 0;
31 }
32 prompt> ./threads 10000
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  volatile int counter = 0;
7  int loops;
8
9  void *worker(void *arg) {
10         int i;
11         for (i = 0; i < loops; i++) {
12                 counter++;
13         }
14         return NULL;
15 }
16
17 int main(int argc, char *argv[]) {
18         if (argc != 2) {
19                 fprintf(stderr, "usage: threads <loops>\n");
20                 exit(1);
21         }
22         loops = atoi(argv[1]);
23         pthread_t p1, p2;
24         printf("Initial value : %d\n", counter);
25         Pthread_create(&p1, NULL, worker, NULL);
26         Pthread_create(&p2, NULL, worker, NULL);
27         Pthread_join(p1, NULL);
28         Pthread_join(p2, NULL);
29         printf("Final value   : %d\n", counter);
30         return 0;
31 }
32 prompt> ./threads 10000
```

Defines the worker function for the threads.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  volatile int counter = 0;
7  int loops;
8
9  void *worker(void *arg) {
10         int i;
11         for (i = 0; i < loops; i++) {
12                 counter++;
13         }
14         return NULL;
15 }
16
17 int main(int argc, char *argv[]) {
18         if (argc != 2) {
19                 fprintf(stderr, "usage: threads <loops>\n");
20                 exit(1);
21         }
22         loops = atoi(argv[1]);
23         pthread_t p1, p2;
24         printf("Initial value : %d\n", counter);
25         Pthread_create(&p1, NULL, worker, NULL);
26         Pthread_create(&p2, NULL, worker, NULL);
27         Pthread_join(p1, NULL);
28         Pthread_join(p2, NULL);
29         printf("Final value   : %d\n", counter);
30         return 0;
31 }
32 prompt> ./threads 10000
```

Main function, entry point of the program.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  volatile int counter = 0;
7  int loops;
8
9  void *worker(void *arg) {
10         int i;
11         for (i = 0; i < loops; i++) {
12                 counter++;
13         }
14         return NULL;
15 }
16
17 int main(int argc, char *argv[]) {
18         if (argc != 2) {
19                 fprintf(stderr, "usage: threads <loops>\n");
20                 exit(1);
21         }
22         loops = atoi(argv[1]);
23         pthread_t p1, p2;
24         printf("Initial value : %d\n", counter);
25         Pthread_create(&p1, NULL, worker, NULL);
26         Pthread_create(&p2, NULL, worker, NULL);
27         Pthread_join(p1, NULL);
28         Pthread_join(p2, NULL);
29         printf("Final value   : %d\n", counter);
30         return 0;
31 }
32 prompt> ./threads 10000
```

Converts the command-line argument to an integer and stores it in ‘loops‘.

Simple Example — Modifying Shared Variable

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  volatile int counter = 0;
7  int loops;
8
9  void *worker(void *arg) {
10         int i;
11         for (i = 0; i < loops; i++) {
12                 counter++;
13         }
14         return NULL;
15 }
16
17 int main(int argc, char *argv[]) {
18         if (argc != 2) {
19                 fprintf(stderr, "usage: threads <loops>\n");
20                 exit(1);
21         }
22         loops = atoi(argv[1]);
23         pthread_t p1, p2;     Declares two thread identifiers.
24         printf("Initial value : %d\n", counter);
25         Pthread_create(&p1, NULL, worker, NULL);
26         Pthread_create(&p2, NULL, worker, NULL);
27         Pthread_join(p1, NULL);
28         Pthread_join(p2, NULL);
29         printf("Final value   : %d\n", counter);
30         return 0;
31 }
32 prompt> ./threads 10000
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  volatile int counter = 0;
7  int loops;
8
9  void *worker(void *arg) {
10         int i;
11         for (i = 0; i < loops; i++) {
12                 counter++;
13         }
14         return NULL;
15 }
16
17 int main(int argc, char *argv[]) {
18         if (argc != 2) {
19                 fprintf(stderr, "usage: threads <loops>\n");
20                 exit(1);
21         }
22         loops = atoi(argv[1]);
23         pthread_t p1, p2;
24         printf("Initial value : %d\n", counter);
25         Pthread_create(&p1, NULL, worker, NULL);
26         Pthread_create(&p2, NULL, worker, NULL);
27         Pthread_join(p1, NULL);
28         Pthread_join(p2, NULL);
29         printf("Final value   : %d\n", counter);
30         return 0;
31 }
32 prompt> ./threads 10000
```

Prints the initial value of the counter.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  volatile int counter = 0;
7  int loops;
8
9  void *worker(void *arg) {
10         int i;
11         for (i = 0; i < loops; i++) {
12                 counter++;
13         }
14         return NULL;
15 }
16
17 int main(int argc, char *argv[]) {
18         if (argc != 2) {
19                 fprintf(stderr, "usage: threads <loops>\n");
20                 exit(1);
21         }
22         loops = atoi(argv[1]);
23         pthread_t p1, p2;
24         printf("Initial value : %d\n", counter);
25         Pthread_create(&p1, NULL, worker, NULL);
26         Pthread_create(&p2, NULL, worker, NULL);
27         Pthread_join(p1, NULL);
28         Pthread_join(p2, NULL);
29         printf("Final value   : %d\n", counter);
30         return 0;
31 }
32 prompt> ./threads 10000
```

Creates the first thread.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  volatile int counter = 0;
7  int loops;
8
9  void *worker(void *arg) {
10         int i;
11         for (i = 0; i < loops; i++) {
12                 counter++;
13         }
14         return NULL;
15 }
16
17 int main(int argc, char *argv[]) {
18         if (argc != 2) {
19                 fprintf(stderr, "usage: threads <loops>\n");
20                 exit(1);
21         }
22         loops = atoi(argv[1]);
23         pthread_t p1, p2;
24         printf("Initial value : %d\n", counter);
25         Pthread_create(&p1, NULL, worker, NULL);
26         Pthread_create(&p2, NULL, worker, NULL);
27         Pthread_join(p1, NULL);
28         Pthread_join(p2, NULL);
29         printf("Final value   : %d\n", counter);
30         return 0;
31 }
32 prompt> ./threads 10000
```

Creates the second thread.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  volatile int counter = 0;
7  int loops;
8
9  void *worker(void *arg) {
10         int i;
11         for (i = 0; i < loops; i++) {
12                 counter++;
13         }
14         return NULL;
15 }
16
17 int main(int argc, char *argv[]) {
18         if (argc != 2) {
19                 fprintf(stderr, "usage: threads <loops>\n");
20                 exit(1);
21         }
22         loops = atoi(argv[1]);
23         pthread_t p1, p2;
24         printf("Initial value : %d\n", counter);
25         Pthread_create(&p1, NULL, worker, NULL);
26         Pthread_create(&p2, NULL, worker, NULL);
27         Pthread_join(p1, NULL);     Waits for the first thread to finish.
28         Pthread_join(p2, NULL);
29         printf("Final value   : %d\n", counter);
30         return 0;
31 }
32 prompt> ./threads 10000
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  volatile int counter = 0;
7  int loops;
8
9  void *worker(void *arg) {
10         int i;
11         for (i = 0; i < loops; i++) {
12                 counter++;
13         }
14         return NULL;
15 }
16
17 int main(int argc, char *argv[]) {
18         if (argc != 2) {
19                 fprintf(stderr, "usage: threads <loops>\n");
20                 exit(1);
21         }
22         loops = atoi(argv[1]);
23         pthread_t p1, p2;
24         printf("Initial value : %d\n", counter);
25         Pthread_create(&p1, NULL, worker, NULL);
26         Pthread_create(&p2, NULL, worker, NULL);
27         Pthread_join(p1, NULL);
28         Pthread_join(p2, NULL);     Waits for the second thread to finish.
29         printf("Final value   : %d\n", counter);
30         return 0;
31 }
32 prompt> ./threads 10000
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  volatile int counter = 0;
7  int loops;
8
9  void *worker(void *arg) {
10         int i;
11         for (i = 0; i < loops; i++) {
12                 counter++;
13         }
14         return NULL;
15 }
16
17 int main(int argc, char *argv[]) {
18         if (argc != 2) {
19                 fprintf(stderr, "usage: threads <loops>\n");
20                 exit(1);
21         }
22         loops = atoi(argv[1]);
23         pthread_t p1, p2;
24         printf("Initial value : %d\n", counter);
25         Pthread_create(&p1, NULL, worker, NULL);
26         Pthread_create(&p2, NULL, worker, NULL);
27         Pthread_join(p1, NULL);
28         Pthread_join(p2, NULL);
29         printf("Final value   : %d\n", counter);
30         return 0;
31 }
32 prompt> ./threads 10000
```

Prints the final value of the counter.

Problems that arise, and must be addressed, when working on many things at once (i.e., concurrently) in the same program.

Details deferred

When there are many concurrently executing **threads** within the same memory space:

- How can we build a correctly working program?
- What primitives are needed from the OS?
- What mechanisms should be provided by the hardware?
- How can we use them to solve the problems of concurrency?

Persistence

- What happens to the data when you switch-off your system?

How to store data persistently?

The **file system** is the part of the OS in charge of managing persistent data.

- What techniques are needed to do so correctly?
- What mechanisms and policies are required to do so with high performance?
- How is reliability achieved, in the face of failures in hardware and software?

Security

Security

- Authentication
- Access Control
- Cryptography

- Convenience
- **Abstraction** of hardware resources for user programs
- Efficiency of usage of CPU, memory, etc.
- Isolation between multiple processes

- Early OSes: Act as a library to provide common functionality across programs
- Later: Go beyond libraries - offer protection
- Evolution from **procedure call** to **system call**.

Procedure call Vs System call

- Can you tell the difference between these?

- When a system call is made to run OS code, the CPU executes at a **higher privilege level**
- The era of multiprogramming - Evolved from running a single program to multiple processes concurrently
- The modern era: Windows/Mac/Linux