Lecture 02: Introduction to Parallel Programming

Vivek Kumar

Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Today's Lecture (Recap of OS Topics)

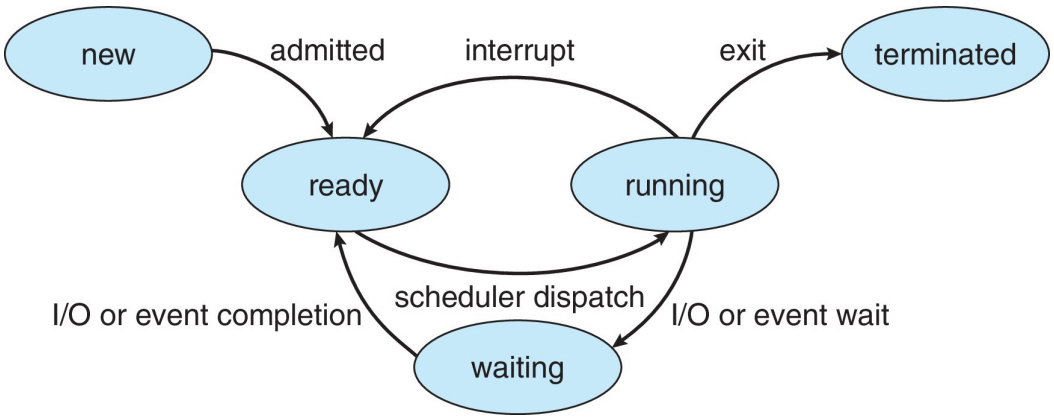
- Processes and threads
- Thread operations
  - Creation and termination
- Mutual exclusion

The Process

```
iiitd@possum:~$ vi fib.c
iiitd@possum:~$ gcc fib.c
iiitd@possum:~$ ./a.out
Fib(40) = 102334155
```

- A program under execution is called a process, and it is a collection of resources
  - Address space (code, heap, stack, etc.)
  - Program counter, registers
  - Open file descriptors
  - CPU affinity, priority, etc.

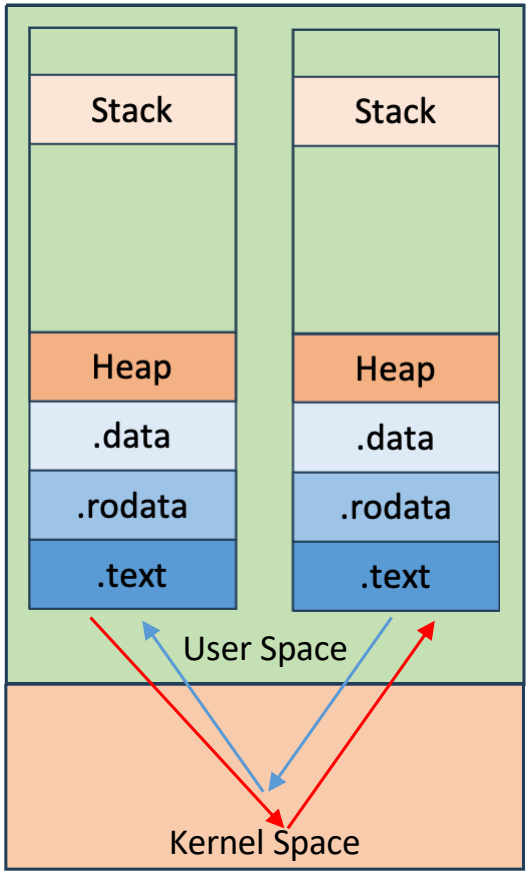
```
  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 2034 iiitd      20   0  4384   820   756 R 100.  0.0  0:17.48 ./a.out
```



- As a process executes, it changes state
  - **New**: The process is being created
    - Loader loading the executable
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution

IPC in a Multicore Processor

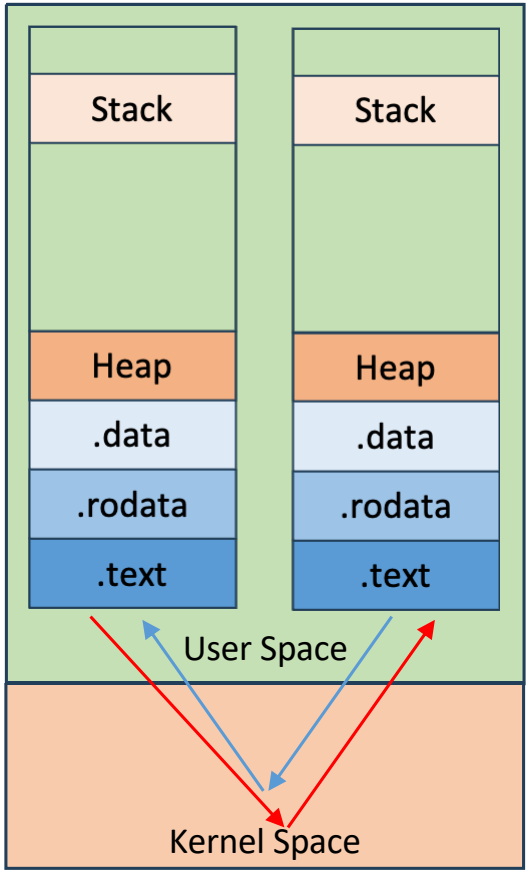
- Inter-process communication in shared memory
  - Transfer of control from user space to kernel space and vice-versa
- Complicated IPC mechanism for communication
- OS has to reserve extra memory / resources
  - Same copy of .text segment in each process
- Separate page table for each process
- Cost of IPC may exceed the cost of actual computation!

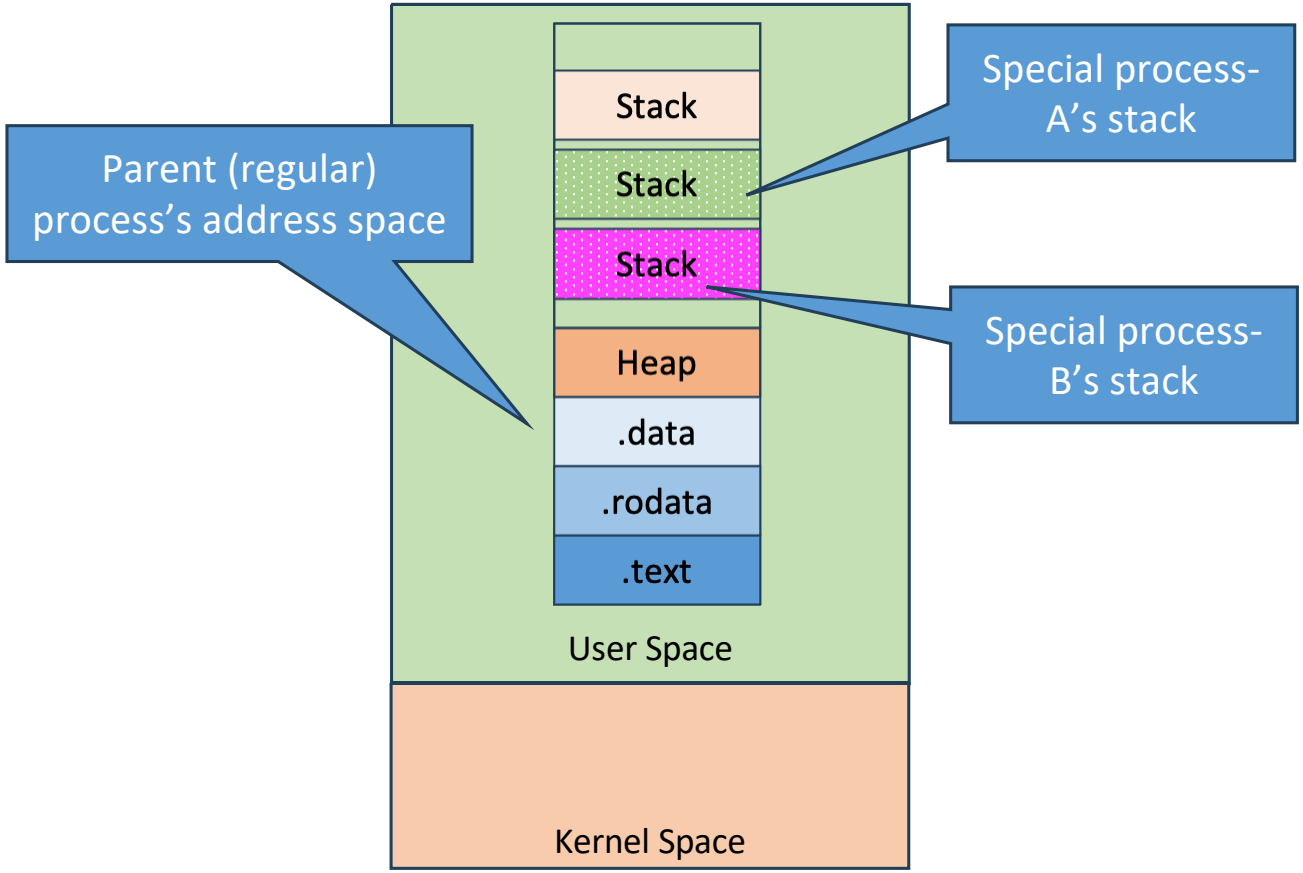


How to Fix the Issues we Highlighted?

- Asking the OS to allow creation of "**special**" processes with **special** powers
- These **special** processes can be created by the parent process, and they all live in harmony like an **ideal** Indian joint family, where entire resources in the house are shared within the family members
  - Sharing of page table of the parent process
  - Sharing of parent's process address space
    - Shared heap, .text, .data segment, etc.
      - Stack cannot be shared as we need each of these "**special**" processes to execute a different method call chain inside the same program
      - Likewise, PC, registers also cannot be shared. Hence, they go through the same set of steps during context switch similar to **regular** processes
  - They can communicate with each other without using any special APIs or without going into the kernel space (zero overheads in communication!)

2 Regular Process v/s 2 Special Process





These “**special**” processes are called as **Threads**!

Thread Creation and Termination in Linux

```
int pthread_create(
  pthread_t *thread,
    //returned identifier for the new thread
  const pthread_attr_t *attr,
    //object to set thread attributes (NULL for default)
  void *(*func)(void *),
    //routine executed after creation
  void *arg
    //a single argument passed to func
) //returns error status
```

```
int pthread_join(
  pthread_t thread,
    //identifier of thread to wait for
  void **status
    //terminating thread's status (NULL to ignore)
) //returns error status
```

Fibonacci Program

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

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

int main(int argc, char *argv[]) {
  uint64_t n = atoi(argv[1]);
  uint64_t result = fib(n);
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

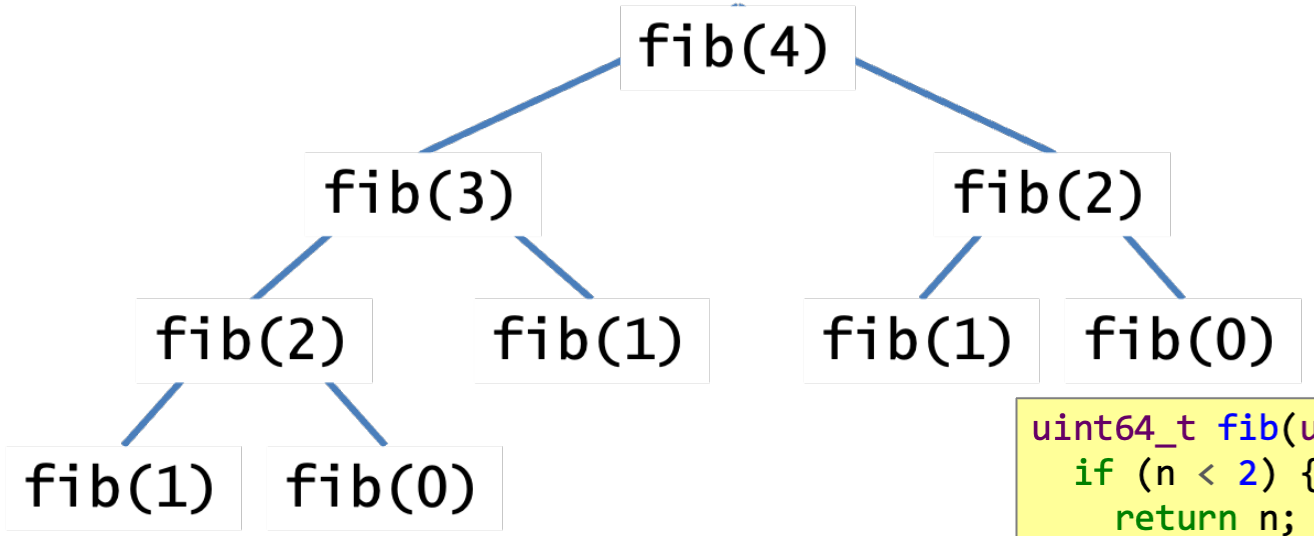
Disclaimer to Algorithms Police

This recursive program is a poor way to compute the nth Fibonacci number, but it provides a good didactic example.

Can we write a parallel version of this code using Pthreads?

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Fibonacci Execution




```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}
```

Key idea for parallelization

The calculations of `fib(n-1)` and `fib(n-2)` can be executed simultaneously without mutual interference.

DAG Source: http://www.cs.ucsb.edu/projects/jicos/tutorial/fibonacci/index.html

Pthread Implementation

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

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

```
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
     result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Pthread Implementation

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Original code

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
     result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Pthread Implementation

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

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Structure for thread arguments

```
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
     result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Pthread Implementation

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Function called when thread is created

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
     result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Pthread Implementation

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

No point in creating thread if there isn't enough to do

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Pthread Implementation

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Marshal input argument to thread

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Pthread Implementation

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Create thread to execute `fib(n-1)`.

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
     result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Pthread Implementation

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Main program executes `fib(n-2)` in parallel.

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
     result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Pthread Implementation

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Block until the auxiliary thread finishes.

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
     result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Pthread Implementation

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Add the results together to produce the final output.

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
     result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Another Implementation

```
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int result = 0;
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  result += fib(i);
  return NULL;
}
```

```
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result += fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Any Issues?

```
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int result = 0;
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  result += fib(i);
  return NULL;
}
```

Race condition !!!

```
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result += fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Mutual Exclusion

- ***Critical section:*** a block of code that access shared modifiable data or resource that should be operated on by only one thread at a time
- ***Mutual exclusion:*** a property that ensures that a critical section is only executed by a thread at a time.
  - *Otherwise it results in a race condition!*
  - Using mutex locks
    - Request lock before executing critical section
    - Enter critical section when lock granted
    - Release lock when leaving critical section

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Pthread Mutex Locks

- Initialize the mutex variable (statically)
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- Lock the mutex
  - `pthread_mutex_lock(&mutex);`
- Unlock the mutex
  - `pthread_mutex_unlock(&mutex);`

Correct Implementation

```
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int result = 0; pthread_mutex_t* mutex=PTHREAD_MUTEX_INITIALIZER;
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  int temp = fib(i); result += temp;
  return NULL;
}
```

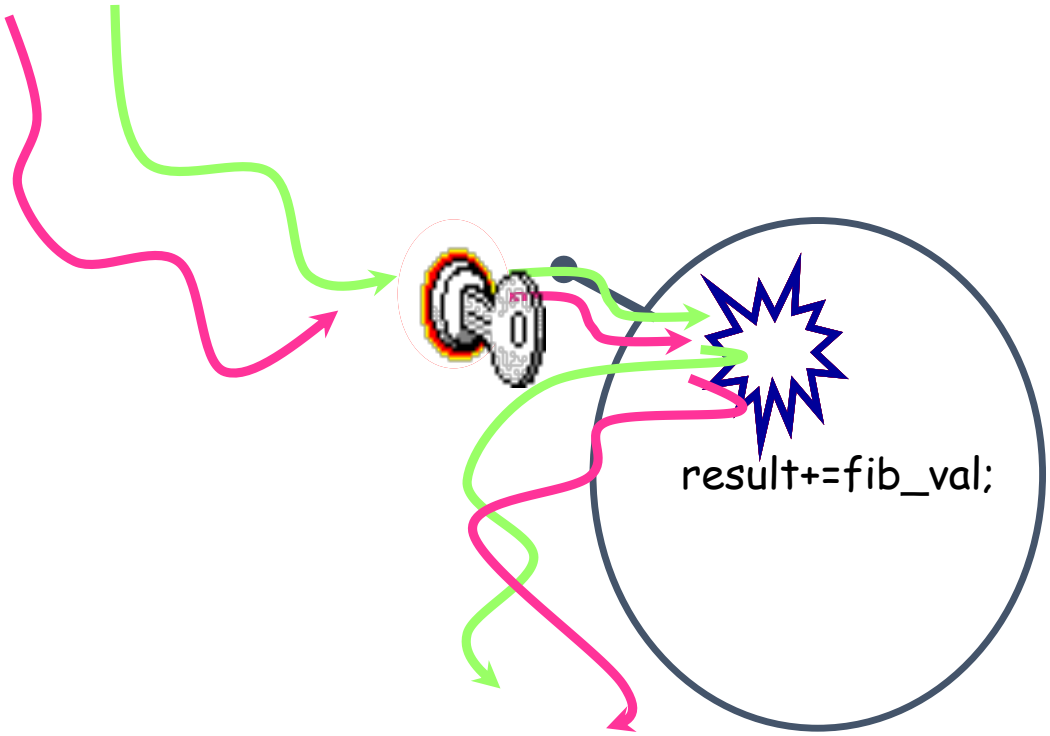
```
pthread_mutex_lock(&mutex);
Critical Section (result += ....)
pthread_mutex_unlock(&mutex);
```

```
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    int temp = fib(n-2); result += temp;
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Visualizing Mutual Exclusion



- Only one **thread** can get the “key” to enter the critical section
- Other **thread(s)** will be queued to get the key

Reminders about this Course!

- No lecture recordings will be provided
- We will not release evaluation rubrics except for quizzes
- We will not release midsem/endsem question paper
- You should learn C/C++ on your own
- We will strictly follow IIITD plagiarism policy

So, plan accordingly. Registering to this course means you are agreeing to all these requirements

Next Lecture

- Concurrency decomposition