Lecture 02: Introduction to Parallel Programming

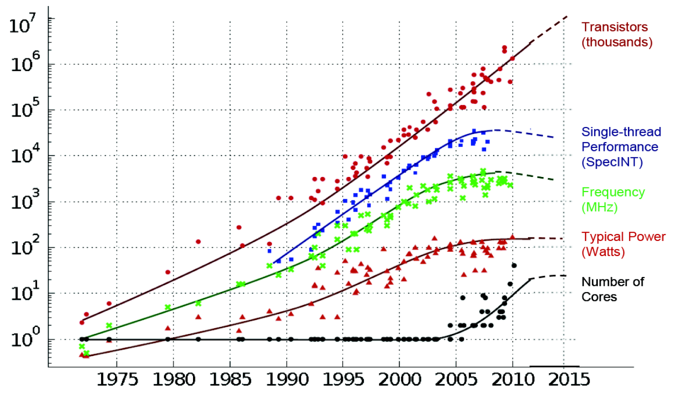
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Today's Lecture

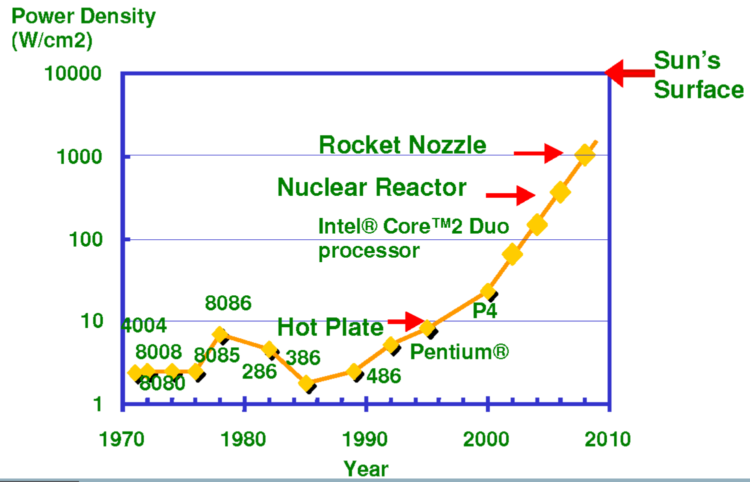
- Processor technology trend
- Thread operations
  - Creation and termination
- Mutual exclusion

Processor Technology Trend

35 YEARS OF MICROPROCESSOR TREND DATA



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore



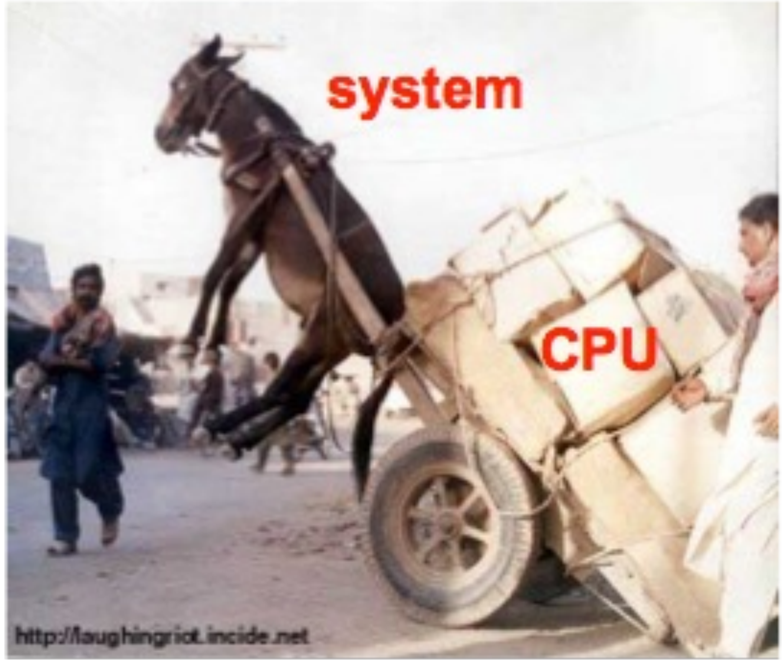
- Moore's law (1964)
  - Area of transistors halves roughly every two years
    - I.e., Total transistors on processor chip gets doubled roughly every two years
- Dennard scaling (1974)
  - Power for fixed chip area remains almost constant even with transistors becoming smaller (power density remains constant)
    - Supply voltage was scaled down proportionally with frequency as transistors shrank
- No more free lunch!
  - Thermal wall hit around 2004
    - Heat dissipation
    - Leakage current
  - Power is proportional to cube of frequency
    - Power = C x V^2 x f and V $\propto$ f
      - Power $\propto$ f^3
    - It restricts frequency growth, but opens up multicore era

Multicore Saves Power

- Nowadays (post Dennard Scaling)
  - Power ~ (Capacitance) * $(\text{Voltage})^2$ * (Frequency)
    - Maximum Frequency is capped by Voltage
  - ***Power is proportional to $(\text{Frequency})^3$***
- Baseline example: single 1GHz core with power P
  - Option A: Increase clock frequency to 2GHz
    - Power = 8P
  - Option B: Use 2 cores at 1 GHz each
    - Power = 2P
- Option B delivers same performance as Option A with 4x less power … provided software can be decomposed to run in parallel !!

Source: https://wiki.rice.edu/confluence/download/attachments/4435861/comp322-s16-lec1-slides.pdf?version=1&modificationDate=1452732285045&api=v2

Adding More Cores Improves performance?

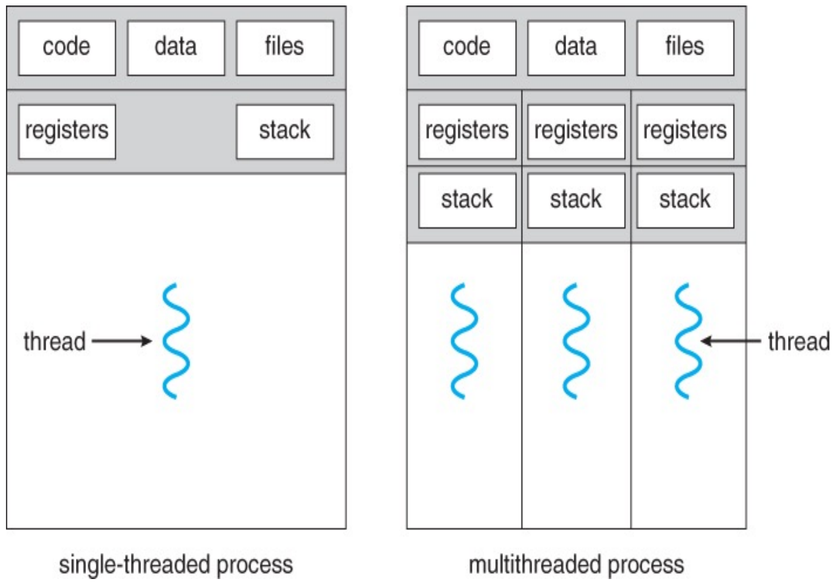


- Computation is just part of the picture
- Memory latency and bandwidth
  - Multiple memory hierarchies with different access latencies (L3, L2, L1, DRAM, Disk)
  - Multicore exacerbates demand
- Inter-processor communication
- Input/Output

Today's Lecture

- Processor technology trend
- Thread operations
  - Creation and termination
- Mutual exclusion

Thread – A Lightweight Process



- Processes are heavyweight
  - Personal address space (allocated memory)
  - Communication across process always requires help from Operating System
- Threads are lightweight
  - Share resources inside the parent process (code, data and files)
    - Easy to communicate across sibling threads!
  - They have their own personal stack (local variables, caller-callee relationship between function)
    - Each thread is assigned a different job in the program
- A process can have one or more threads

Thread Creation in Linux

```
//Asynchronously invoke func in a new thread
int pthread_create(
              //returned identifier for the new thread
              pthread_t *thread,

              //specifies the size of thread's stack and
              //how the thread should be scheduled by OS
              const pthread_attr_t *attr,

              //routine executed after creation
              void *(*func)(void *),

              //a single argument passed to func
              void *arg
) //returns error status
```

Waiting for Thread Termination in Linux

```
//Suspend execution of calling thread until thread
//terminates
int pthread_join(
  //identifier of thread to wait for
              pthread_t thread,

  //terminating thread’s status (NULL to ignore)
              void **status
) //returns error status
```

Fibonacci Program

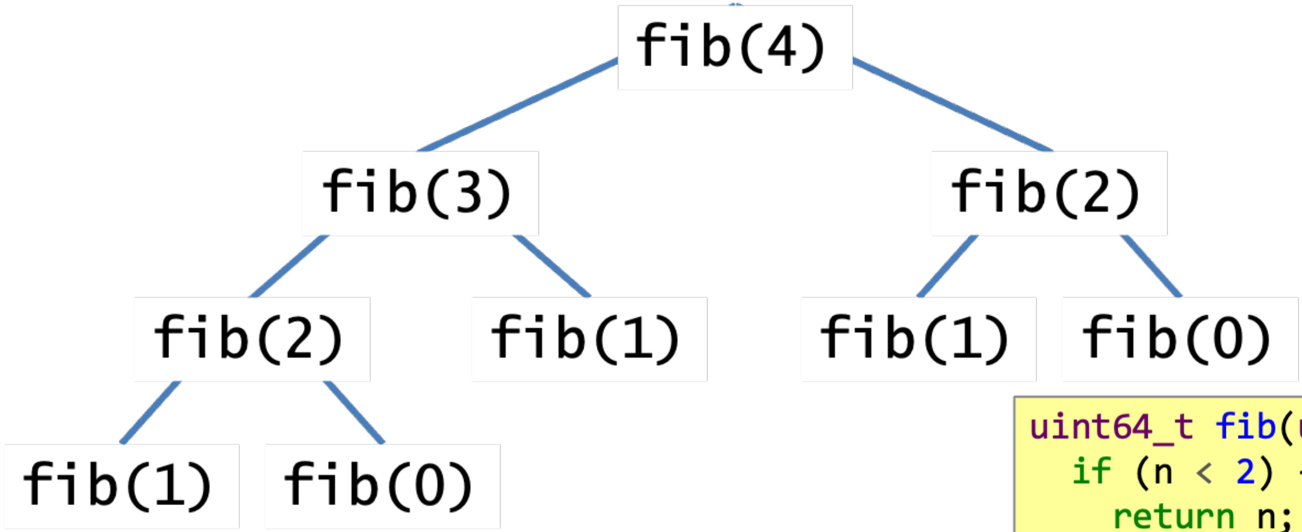
```
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>

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 of Fibonacci

```
#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 of Fibonacci

```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 of Fibonacci

```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;
}
```

Structure for thread arguments

```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 of Fibonacci

```
#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

```
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 of Fibonacci

```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 of Fibonacci

```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],
  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 of Fibonacci

```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;
}
```

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

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

Pthread Implementation of Fibonacci

```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;
}
```

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

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

Pthread Implementation of Fibonacci

```
#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.

```
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 of Fibonacci

```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;
}
```

Add the results together to produce the final output.

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

Today's Lecture

- Processor technology trend
- Thread operations
  - Creation and termination
- Mutual exclusion

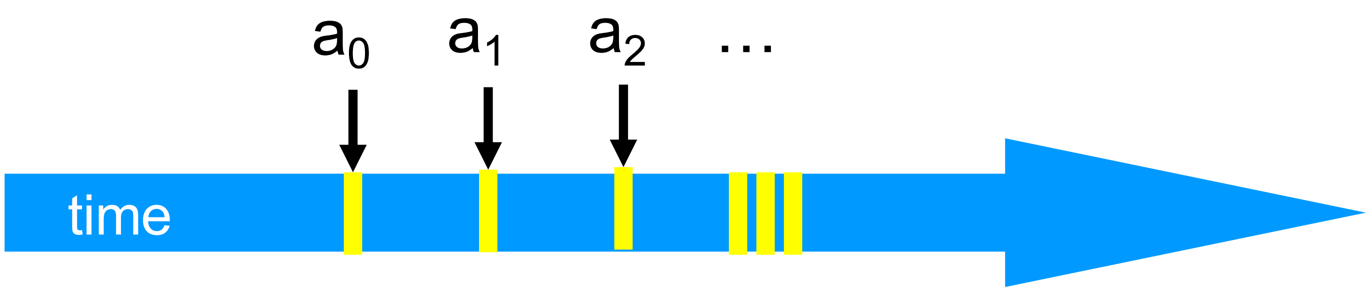
Mutual Exclusion

- ***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!*

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

Threads

- A ***thread*** A is (formally) a sequence a_0, a_1, ... of events
  - Notation: $a_0 \rightarrow a_1$ indicates order



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

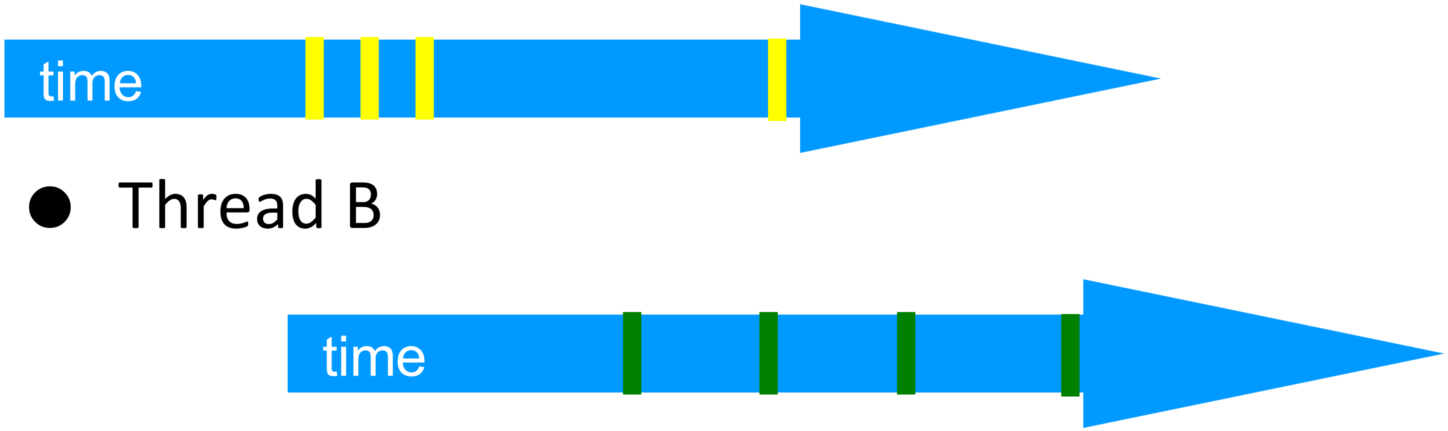
Example Thread Events

- Assign to shared variable
- Assign to local variable
- Invoke method
- Return from method
- Lots of other things …

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

Concurrency

- Thread A
- Thread B



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

Interleavings

- Events of two or more threads
  - Interleaved
  - Not necessarily independent (why?)

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

Critical Sections and Mutual Exclusion

- Critical section is the code (block of code) that should be executed by only one thread at a time
- **Mutex locks** enforce mutual exclusion in Pthreads
  - Mutex lock states: locked and unlocked
  - Only one thread can lock a mutex lock at any particular time
- Using mutex locks
  - Request lock before executing critical section
  - Enter critical section when lock granted
  - Release lock when leaving critical section

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);`

Condition Variables for Synchronization

- Using a condition variable
  - thread can block itself until a condition becomes true
    - thread locks a mutex
    - tests a predicate defined on a shared variable

      if predicate is false, then wait on the condition variable

      waiting on condition variable unlocks associated mutex
  - when some thread makes a predicate true
    - that thread can signal the condition variable to either

      wake one waiting thread

      wake all waiting threads
    - when thread releases the mutex, it is passed to first waiter

Source: https://www.clear.rice.edu/comp422/lecture-notes/comp422-2016-Lecture8-Pthreads.pdf

Pthread Condition Variable APIs

```
/* initialize or destroy a condition variable */
int pthread_cond_init(pthread_cond_t *cond,
                                       const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);

/* block until a condition is true */
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_ *mutex);

/* signal one or all waiting threads that condition  is true */
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

wake one

wake all

Source: https://www.clear.rice.edu/comp422/lecture-notes/comp422-2016-Lecture8-Pthreads.pdf

Wait/Notify Sequence in Pthread

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

Consumer(s)

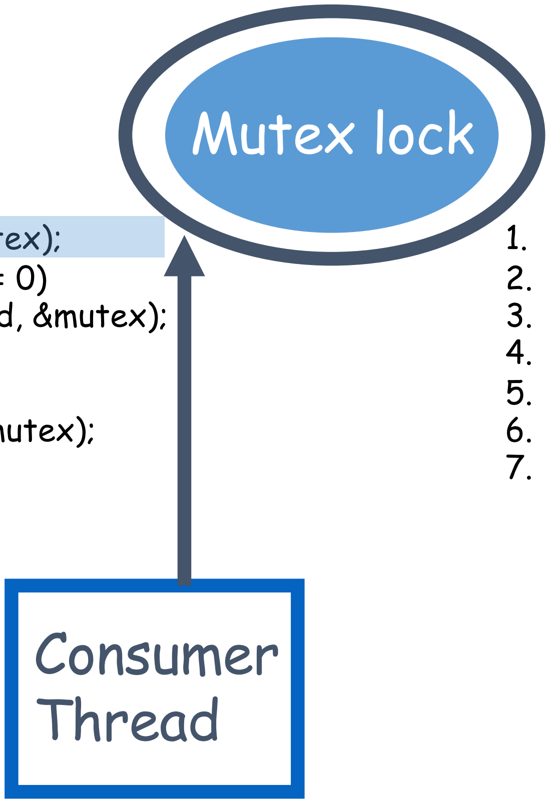
```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Producer

© Vivek Kumar

Wait/Notify Sequence in Pthread

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```




```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```


© Vivek Kumar

Wait/Notify Sequence in Pthread

Mutex lock

```
1.  pthread_mutex_lock(&mutex);
2.  while(task_queue_size() == 0)
3.    pthread_cond_wait(&cond, &mutex);
4.  }
5.  task = pop_task_queue();
6.  pthread_mutex_unlock(&mutex);
7.  execute_task (task);
```

```
1.  pthread_mutex_lock(&mutex);
2.  int queue_size = task_queue_size();
3.  push_task_queue(&task);
4.  if(queue_size == 0) {
5.    pthread_cond_broadcast(&cond);
6.  }
7.  pthread_mutex_unlock(&mutex);
```

Consumer Thread

© Vivek Kumar

Wait/Notify Sequence in Pthread

Mutex lock

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

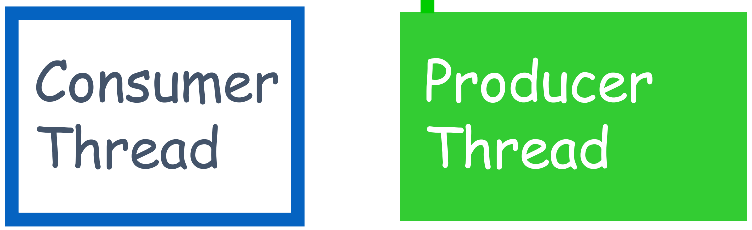
Consumer Thread

© Vivek Kumar

Wait/Notify Sequence in Pthread


```
1.  pthread_mutex_lock(&mutex);
2.  while(task_queue_size() == 0)
3.    pthread_cond_wait(&cond, &mutex);
4.  }
5.  task = pop_task_queue();
6.  pthread_mutex_unlock(&mutex);
7.  execute_task (task);
```

```
1.  pthread_mutex_lock(&mutex);
2.  int queue_size = task_queue_size();
3.  push_task_queue(&task);
4.  if(queue_size == 0) {
5.    pthread_cond_broadcast(&cond);
6.  }
7.  pthread_mutex_unlock(&mutex);
```



© Vivek Kumar

Wait/Notify Sequence in Pthread

Mutex lock

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Consumer Thread

Producer Thread

© Vivek Kumar

Wait/Notify Sequence in Pthread

Mutex lock

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Consumer Thread

Producer Thread

© Vivek Kumar

Wait/Notify Sequence in Pthread

Mutex lock

1. pthread_mutex_lock(&mutex);
2. while(task_queue_size() == 0)
3.   pthread_cond_wait(&cond, &mutex);
4. }
5. task = pop_task_queue();
6. pthread_mutex_unlock(&mutex);
7. execute_task (task);

1. pthread_mutex_lock(&mutex);
2. int queue_size = task_queue_size();
3. push_task_queue(&task);
4. if(queue_size == 0) {
5.   pthread_cond_broadcast(&cond);
6. }
7. pthread_mutex_unlock(&mutex);

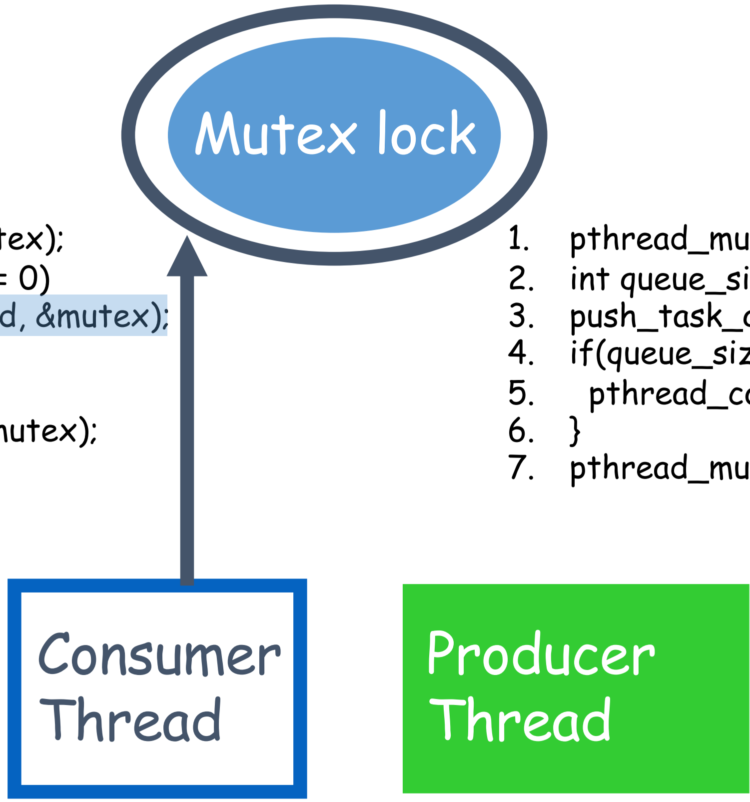
Consumer Thread

Producer Thread

© Vivek Kumar

Wait/Notify Sequence in Pthread

```
1.  pthread_mutex_lock(&mutex);
2.  while(task_queue_size() == 0)
3.    pthread_cond_wait(&cond, &mutex);
4.  }
5.  task = pop_task_queue();
6.  pthread_mutex_unlock(&mutex);
7.  execute_task (task);
```




```
1.  pthread_mutex_lock(&mutex);
2.  int queue_size = task_queue_size();
3.  push_task_queue(&task);
4.  if(queue_size == 0) {
5.    pthread_cond_broadcast(&cond);
6.  }
7.  pthread_mutex_unlock(&mutex);
```


© Vivek Kumar

Wait/Notify Sequence in Pthread

Mutex lock

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Consumer Thread

Producer Thread

© Vivek Kumar

Wait/Notify Sequence in Pthread

Mutex lock

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Consumer Thread

Producer Thread

© Vivek Kumar

Reminders about this Course!

- No lecture recordings will be provided
- No help will be provided on project group formation
- You should not open-source the course projects after the course is over
- 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 03

- Productivity in parallel programming