CSE502: Foundations of Parallel Programming

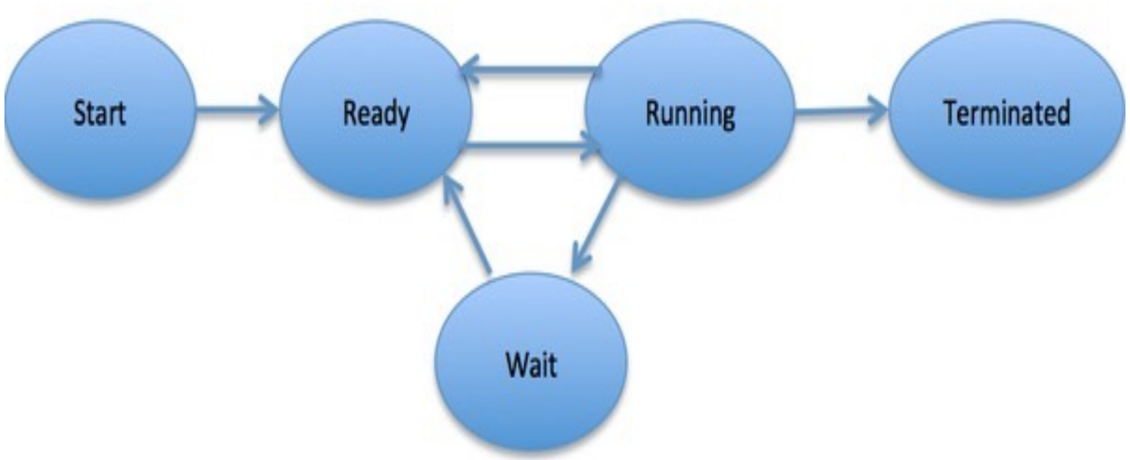
Lecture 02: Refresher – Processes and Threads

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

Today's Class

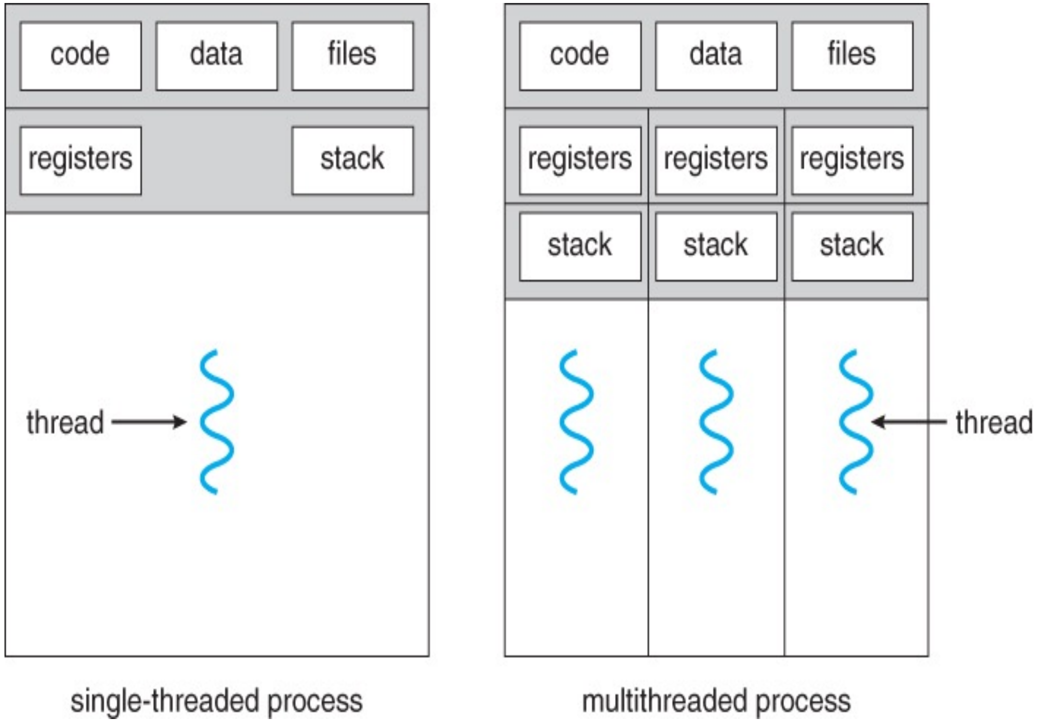
- Processes and threads
- Shared memory parallel programming using Pthreads
  - Pthread creation and joining
  - Critical sections and mutual exclusion
  - Condition variables for synchronizations

Process is a Program in Execution



- Process contains:
  - Program instructions (code)
  - Program data (global variables)
  - Program counter (address of the currently executing instruction)
  - CPU registers
  - Stack (local variables, caller-callee relationship between function)
- Diagram on left shows process life-cycle
  - New – process being created
  - Ready – waiting for a free processor
  - Running – instructions are executing
  - Waiting – waiting for some event (I/O, etc.)
  - Terminated – execution is completed

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

Advantages of Multithreading

- Responsiveness
  - Even if part of program is blocked or performing lengthy operation, multithreading allows the program to continue
- Economical resource sharing
  - Threads share memory and resources of their parent process which allows multiple tasks to be performed simultaneously inside the process
- Utilization of multicores
  - Easily scale on modern multicore processors

POSIX Thread API (Pthreads)

- Standard threads API supported on almost all platforms
- Do-it-yourself scheduling (tasks-to-threads mapping)
- Each thread implements an abstraction of a processor, which are multiplexed onto machine resources
- Threads communicate though shared memory
  – Very cheap than inter-process communication

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

Why Should I Care About Pthreads?

Pthreads is the foundation for multithreaded programming models

- Used to implement several parallel programming models, such as OpenMP, Cilk, X10, TBB, Habanero-C, etc.
- **You will have a hard time understanding this course without a background on Pthreads**

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

Key Pthread APIs

- Thread creation and joining
- Critical section and mutual exclusion
- Condition variables for synchronization

Pthread Creation

```
//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 managed 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
```

Wait for Pthread Termination

```
//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

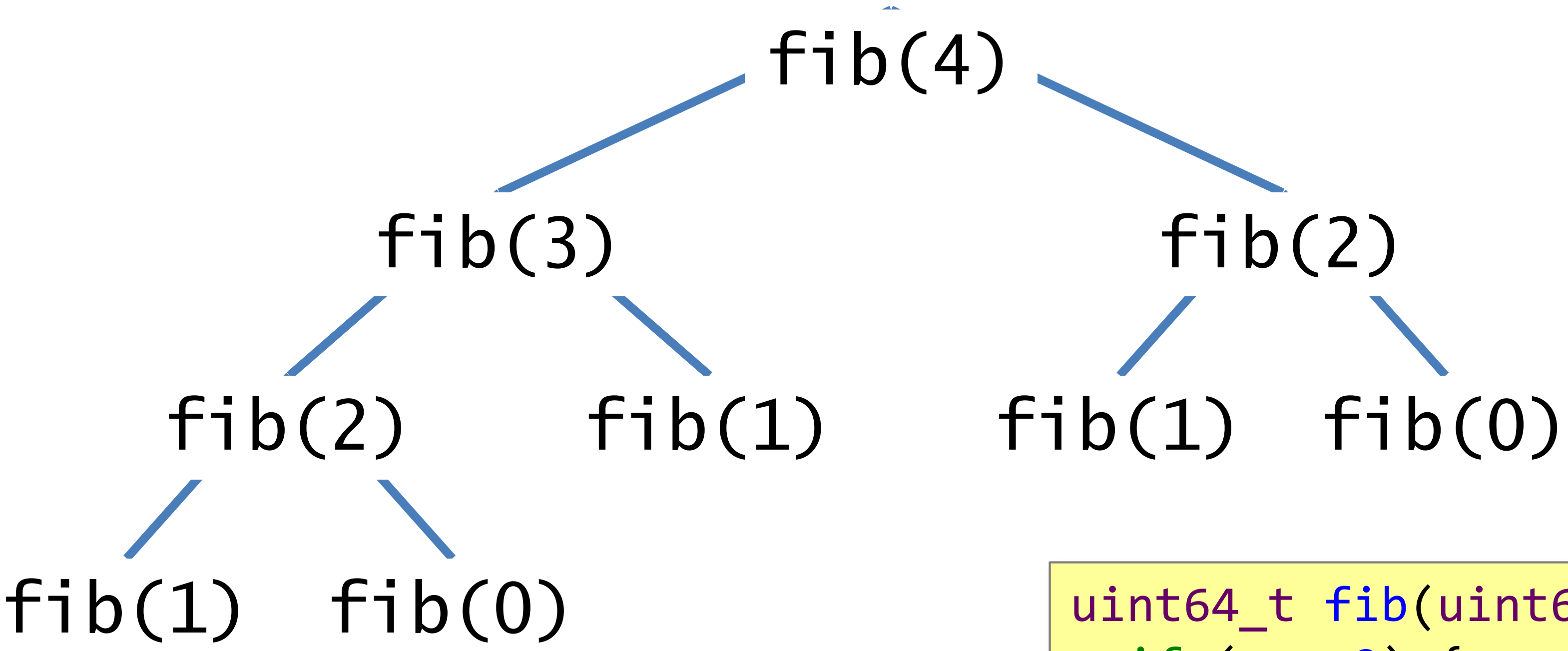
```c
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>

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

```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;
}
```


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;
}
```

Original code.

```
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;
}
```

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 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;
}
```

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 of Fibonacci

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

uint64_t fib(uint
  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;
}
```

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

```
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
  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

```
#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;
}
```

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(v
  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 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;
}
```

Block until the auxiliary thread finishes.

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 Class

- Shared memory parallel programming using Pthreads
  - Pthread creation and joining
  - Critical sections and mutual exclusion
  - Condition variables for synchronizations

Critical Sections and Mutual Exclusion

- Critical section = code executed by only one thread at a time

```
/* threads compete to update global variable minval */
if (my_minval < minval)
    minval = my_minval;
```

- 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
- Operations

created by **pthread_mutex_attr_init** specifies mutex type

```
int pthread_mutex_init(pthread_mutex_t *mutex_lock,
        const pthread_mutexattr_t *lock_attr)
int pthread_mutex_lock(pthread_mutex_t *mutex_lock)
int pthread_mutex_unlock(pthread_mutex_t *mutex_lock)
```


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

Reduction Using Mutex Locks

```
pthread_mutex_t _lock;
int minval;
...
int main() {
    ...
    pthread_mutex_init(&_lock, NULL);
    ...
}
void *find_minval(void *list_ptr) {
    ...
    pthread_mutex_lock(&_lock);   /* lock the mutex */
    if (my_minval < minval)
    minval = my_minval;
    pthread_mutex_unlock(&_lock); /* unlock the mutex */
}
```

Critical Section

Today's Class

- Shared memory parallel programming using Pthreads
  - Pthread creation and joining
  - Critical sections and mutual exclusion
  - Condition variables for synchronizations

Condition Variables for Synchronization

Condition variable: associated with a predicate and a mutex

- 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_t *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);
```

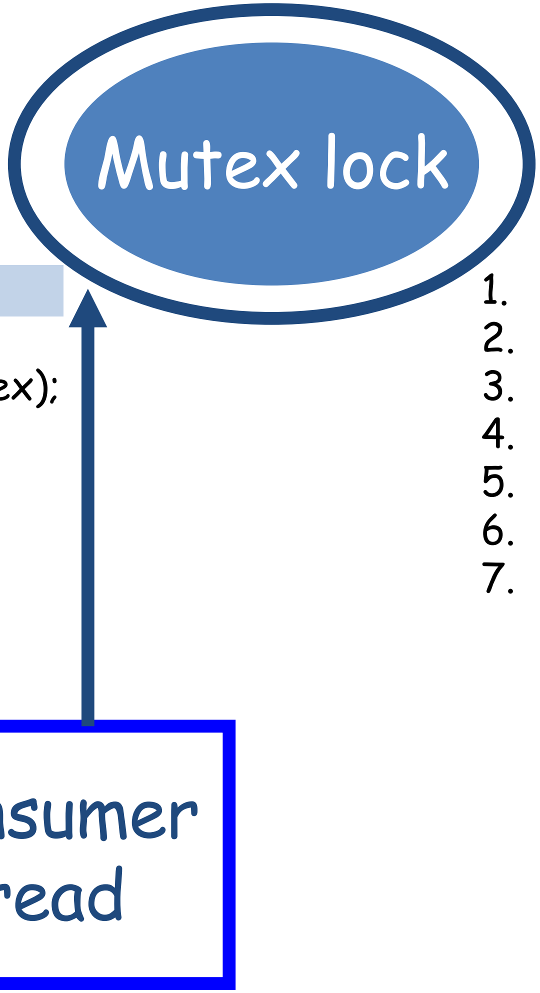
Consumer(s)

```
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

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);
```

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

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);
```

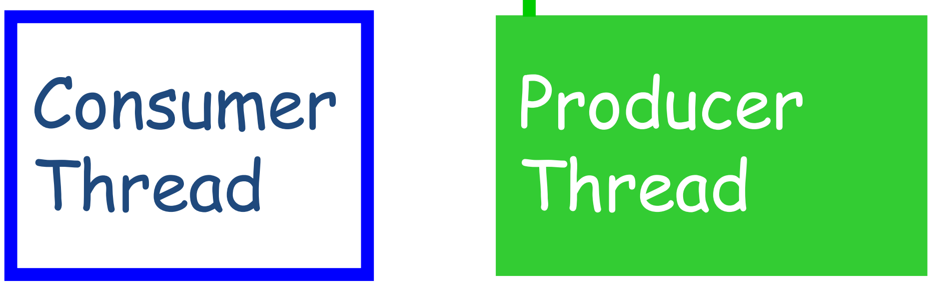
```
1.  pthread_mutex_lock(&mutex);
2.  int queue_size = task_queue_size();
3.  push_task_queue(&task);
4.  if(queue_size == 0) {
5.    pthread_cond_broadcast(&cond);
6.  }
7.  pthread_mutex_unlock(&mutex);
```

Consumer Thread

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);
```



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

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

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);
```

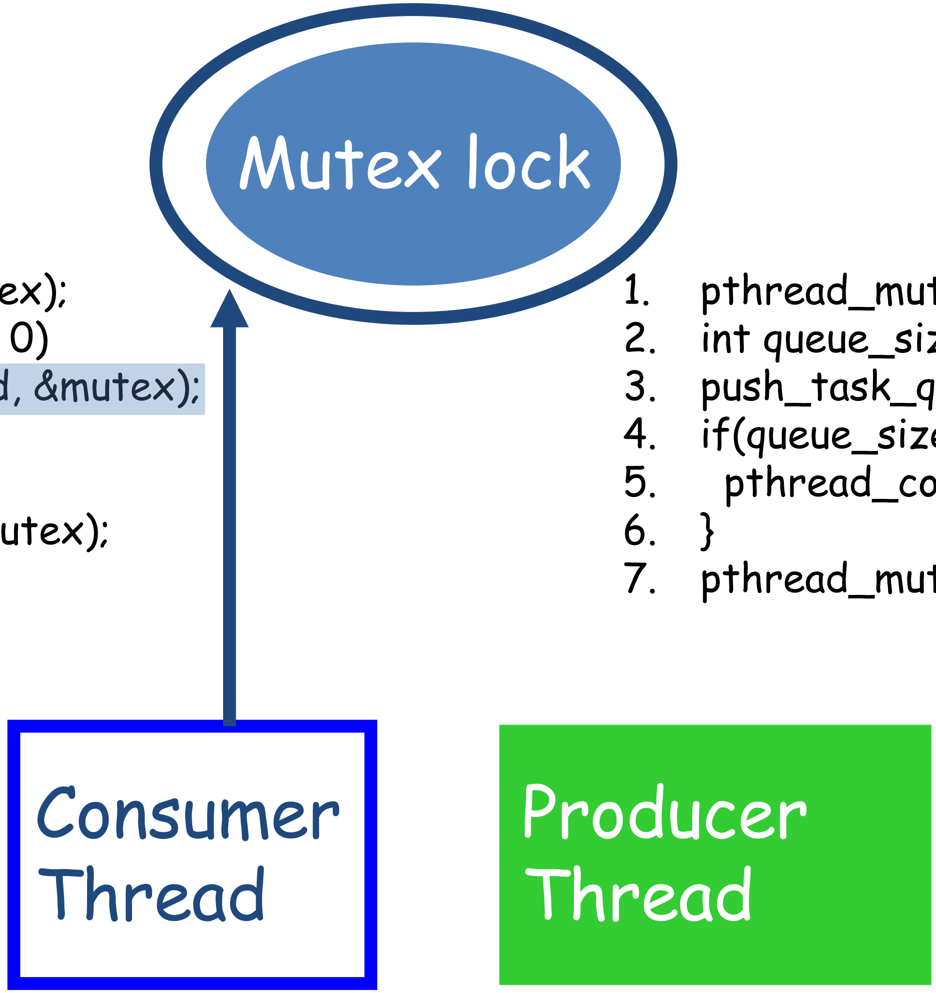
```
1.  pthread_mutex_lock(&mutex);
2.  int queue_size = task_queue_size();
3.  push_task_queue(&task);
4.  if(queue_size == 0) {
5.    pthread_cond_broadcast(&cond);
6.  }
7.  pthread_mutex_unlock(&mutex);
```

Consumer Thread

Producer Thread

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);

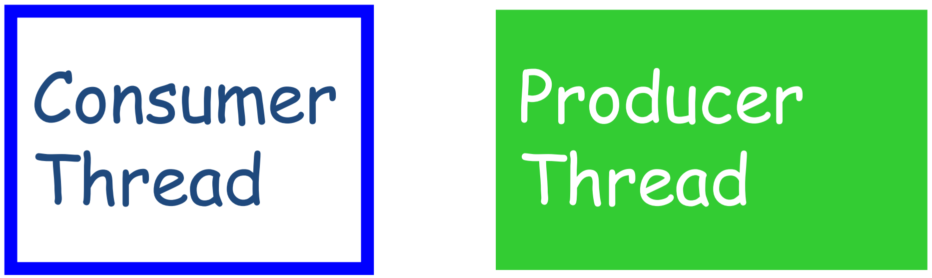


1. pthread_mutex_lock(&mutex);
2. int queue_size = task_queue_size();
3. push_task_queue(&task);
4. if(queue_size == 0) {
5.   pthread_cond_broadcast(&cond);
6. }
7. pthread_mutex_unlock(&mutex);

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);
```



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

Reminders about this Course!!

- **No** lecture recordings
- You are **not** allowed to open-source the course assignments/labs/projects even 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

Reading Materials

- Process and threads
  - Please go though your favorite Operating Systems book and read the chapters on processes and threads
- POSIX Threads programming
  - https://hpc-tutorials.llnl.gov/posix/
  - Note: Pthread APIs related to thread-specific data were not discussed in class and you should read it on your own. There are plenty of online resources
    - pthread_key_create, pthread_setspecific, pthread_getspecific

Next Class

- Introduction to parallel architectures and programming models
- Assignment-1 will be announced on **21/01** noon with a deadline of **24/01** midnight
  - No extensions!

Acknowledgements

- Several of the slides used in this course are borrowed from the following online course materials:
  - Course COMP322, Prof. Vivek Sarkar, Rice University
  - Course COMP 422, Prof. John Mellor-Crummey, Rice University
  - Course CSE539S, Prof. I-Ting Angelina Lee, Washington University in St. Louis
- Contents are also borrowed from following sources:
  - "Introduction to Parallel Computing" by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003
  - https://computing.llnl.gov/tutorials/parallel_comp/
  - https://images.google.com/