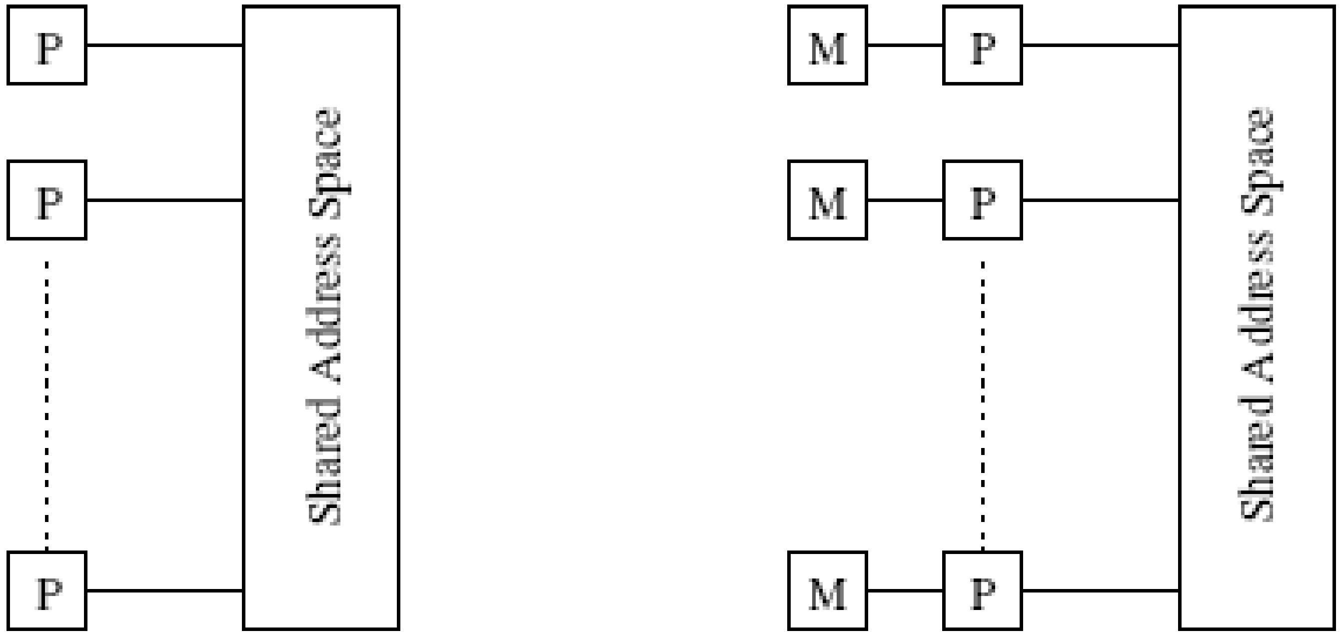
Programming Shared Address Space Platforms (Pthreads)

Thread Basics

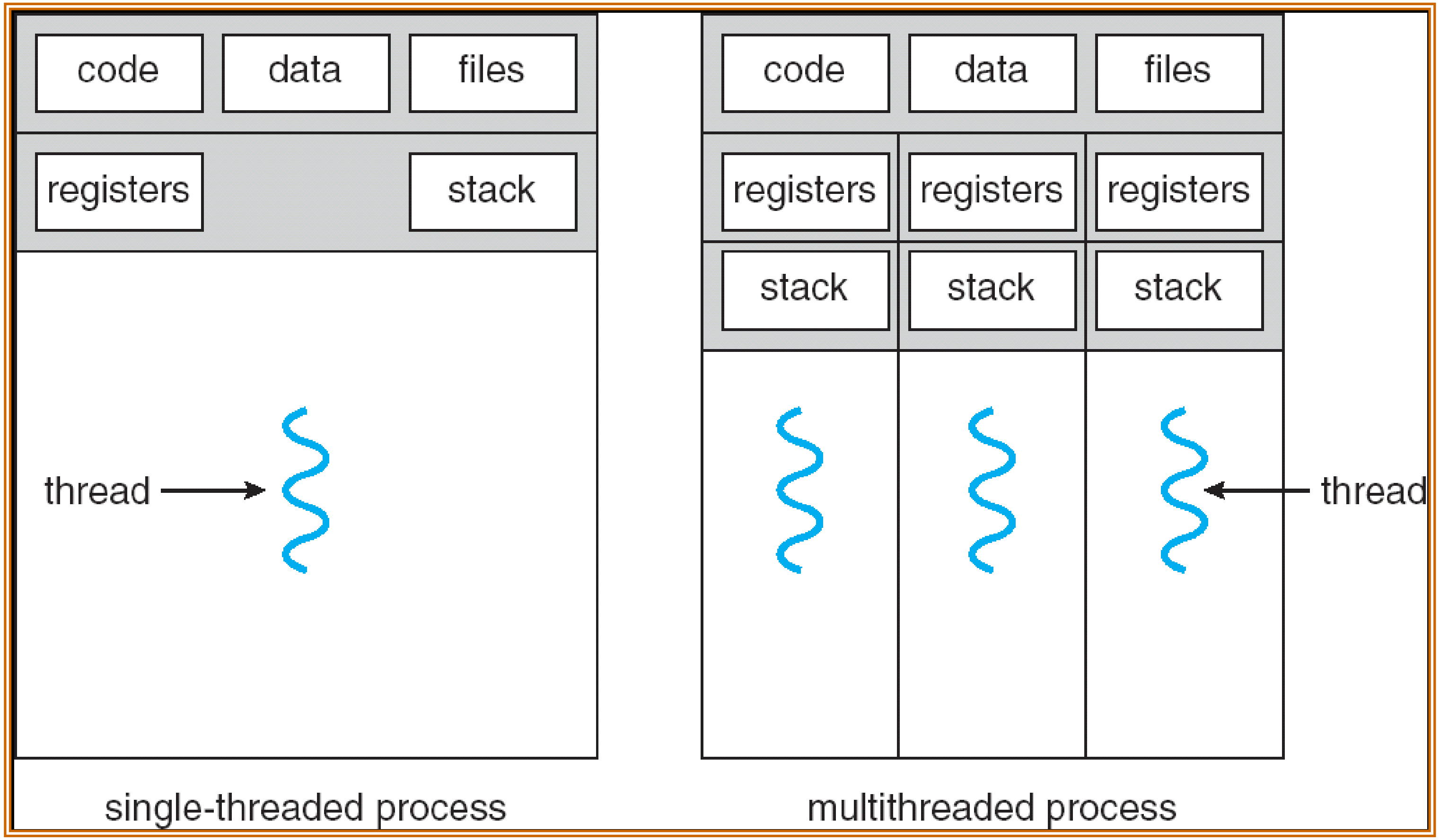


The logical machine model of a thread-based parallel programming paradigm.

Thread Basics

- All memory in the logical machine model of a thread is globally accessible to every other thread.
- The stack corresponding to the function call is generally treated as being local to the thread for liveness reasons.

Thread Basics



Thread Basics

- Cannot assume any order of execution.
- Any such order must be explicitly established using synchronization mechanisms.
- Issues on deadlock, starvation and performance.

Pthreads API

- The Pthreads API is defined in the ANSI/IEEE POSIX 1003.1 - 1995 standard.
- The subroutines which comprise the Pthreads API can be informally grouped into three major classes:
  - **Thread management**
  - **Mutexes**
  - **Condition variables**

Thread Management

- The first class of functions work directly on threads - creating, detaching, joining, etc.
- They also include functions to set/query thread attributes (joinable, scheduling etc.)

Creating Threads

- Initially, only a single, default thread. All other threads must be explicitly created by the programmer.
- **routine:** pthread_create(*thread,attr,start_routine,arg*)
  - **thread:** unique identifier for the new thread (*pthread_t*)
  - ***attr***: attribute object used to set thread attributes. (*pthread_attr*) You can specify a thread attributes object, or NULL for the default values.
  - ***start_routine***: C routine that the thread will execute.
  - ***arg***: A single argument that may be passed to *start_routine*. It must be passed by reference. NULL may be used if no argument is to be passed.
- If successful, the pthread_create() function shall return zero; otherwise, an error number shall be returned to indicate the error.

Thread Attributes

- By default, a thread is created with certain attributes. Some of these attributes can be changed by the programmer via the thread attribute object.
- pthread_attr_init(*attr*) and pthread_attr_destroy(*attr*) are used to initialize/destroy the thread attribute object.
- Other routines are then used to query/set specific attributes in the thread attribute object.

Terminating Thread

- There are several ways in which a Pthread may be terminated:
  - The thread makes a call to the pthread_exit() subroutine.
  - The thread is cancelled by another thread via the pthread_cancel() routine.
  - The entire process is terminated due to a call to the exit subroutine.

Terminating Thread

- **routine:** pthread_exit(*status*)
- used to explicitly exit a thread.
- The programmer may optionally specify a termination status, which is stored as a void pointer for any thread that may join the calling thread.
- Cleanup: the pthread_exit() routine does not close files; any files opened inside the thread will remain open after the thread is terminated.

Example

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid) {
    int *tid;
    tid = (int *) threadid;
    printf("Hello World! It's me, thread #%d!\n", *tid);
    pthread_exit(NULL);
}
```

Example

```
int main(int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   int rc, t, tids[NUM_THREADS];
   for(t=0;t<NUM_THREADS;t++){
      printf("In main: creating thread %d\n", t);
      tids[t] = t;
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &tids[t]);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }

   pthread_exit(NULL);
}
```

Passing Arguments to Threads

- The pthread_create() routine permits the programmer to pass one argument to the thread start routine.
- For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the pthread_create() routine.
- All arguments must be passed by reference and cast to (void *).

Passing Arguments to Threads

```
struct two_args {
    int arg1;
    int arg2;
};
void *needs_2_args(void *ap) {
    struct two_args *argp;
    int a1, a2;
    …
    argp = (struct two_args *) ap;
    a1 = argp->arg1;
    a2 = argp->arg2;
    …
    free (argp);
    pthread_exit(NULL);
}
```

Passing Arguments to Threads

```
int main(int argc, char *argv[]) {
    pthread_t t;
    struct two_args *ap;
    int rc;
    …
    ap = (struct two_args *) malloc (sizeof (struct two_args));
    ap->arg1 = 1;
    ap->arg2 = 2;
    rc = pthread_create(&t, NULL, needs_2_args, (void *) ap);
    …
    pthread_exit(NULL);
}
```

Joining & Detaching Threads

- Routines:
  pthread_join (threadid,status)
  pthread_detach (threadid,status)
  pthread_attr_setdatachstate (attr,detachstate)
  pthread_attr_getdetachstate (attr,datachstate)

- "Joining" is one way to accomplish synchronization between threads.
- The pthread_join() subroutine blocks the calling thread until the specified threadid thread terminates.
- The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit().

Joining & Detaching Threads

- When a thread is created, one of its attributes defines whether it is joinable or detached.
- Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
- To explicitly create a thread as joinable or detached, the attr argument in the pthread_create() routine is used:
  - Declare a pthread attribute variable of the pthread_attr_t data type
  - Initialize the attribute variable with pthread_attr_init()
  - Set the attribute detached status with pthread_attr_setdetachstate()
  - When done, free library resources used by the attribute with pthread_attr_destroy()

Example

```
void *BusyWork(void *null) {
    …
    pthread_exit((void *) 0);
}
```

Example

```
int main (int argc, char *argv[]) {
    pthread_attr_t attr;
    int rc, t;
    void *status;
    /* Initialize and set thread detached attribute */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    …
    /* Free attribute and wait for the other threads */
    pthread_attr_destroy(&attr);
    for(t=0; t<NUM_THREADS; t++) {
        rc = pthread_join(thread[t], &status);
        …
        printf("Completed join with thread %d status= %ld\n",t,
    (long)status);
    }
    pthread_exit(NULL);
}
```

Synchronization Issues

- When multiple threads attempt to manipulate the same data item, the results can often be incoherent if proper care is not taken to synchronize them.
- Consider:

```
/* each thread tries to update variable best_cost as
   follows */
if (my_cost < best_cost)
     best_cost = my_cost;
```

- Assume that there are two threads, the initial value of best_cost is 100, and the values of my_cost are 50 and 75 at threads t1 and t2.
- Depending on the schedule of the threads, the value of best_cost could be 50 or 75 - A race condition problem!

Race Condition

- A race condition occurs when
  - Multiple processes or threads read and write shared data items
  - They do so in a way where the final result depends on the order of execution of the processes.
- The output depends on who finishes the race last.

- Cannot assume any order of execution.
- Any such order must be explicitly established using synchronization mechanisms.

Mutex

- The second class of functions deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion".
- Mutex functions provide for creating, destroying, locking and unlocking mutexes.
- They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
- When a process, or thread executes code that manipulates shared data, we say it is in its Critical Section.
- A general structure for critical section:

```
…
lock mutex
    critical section
unlock mutex
    noncritical section
```

Creating & Destroying Mutex

- Mutex must be declared with type pthread_mutex_t, and must be initialized before they can be used.
- There are two ways to initialize a mutex variable:
  - Statically, when it is declared. For example:

  pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
  - Dynamically, with the pthread_mutex_init() routine. This method permits setting mutex object attributes, *attr* (which may be specified as NULL to accept defaults).
- **Routines:**

  pthread_mutex_init (*mutex,attr*)
  pthread_mutex_destroy (*mutex*)
  pthread_mutexattr_init (*attr*)
  pthread_mutexattr_destroy (*attr*)
- The mutex is initially unlocked.

Locking & Unlocking Mutex

- **Routines:**
  pthread_mutex_lock (*mutex*)
  pthread_mutex_unlock (*mutex*)
  pthread_mutex_trylock (*mutex*)

- pthread_mutex_lock() is used by a thread to acquire a lock on the specified *mutex* variable.
- pthread_mutex_unlock() will unlock a mutex if called by the owning thread. An error will be returned if:
  - the mutex was already unlocked
  - the mutex is owned by another thread
- pthread_mutex_trylock() will attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately with an "EBUSY" error code. This routine may be useful in preventing deadlock conditions.

Example 1

- We can now write our previously incorrect code segment as:
- ```
  ....
  pthread_mutex_t minimum_value_lock;
  int best_cost; //global variable!
  main() {
      ....
      pthread_mutex_init(&minimum_value_lock, NULL);
      ....
  }
  void *find_min(void *list_ptr) {
      ....
      pthread_mutex_lock(&minimum_value_lock);
      if (my_cost < best_cost)
          best_cost = my_cost;
      pthread_mutex_unlock(&minimum_value_lock);
  }
  ```
  ```

Example 2

- The producer-consumer scenario imposes the following constraints:
  - The producer thread must not overwrite the shared buffer when the previous task has not been picked up by a consumer thread.
  - The consumer threads must not pick up tasks until there is something present in the shared data structure.
  - Individual consumer threads should pick up tasks one at a time.

Example 2

```
pthread_mutex_t task_queue_lock;
int task_available;
...
main() {
    ....
    task_available = 0;
    pthread_mutex_init(&task_queue_lock, NULL);
    ....
}
```

Example 2

```
void *producer(void *producer_thread_data) {
    ....
    while (!done()) {
        inserted = 0;
        create_task(&my_task);
        while (inserted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 0) {
                insert_into_queue(my_task);
                task_available = 1;
                inserted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
    }
}
```

Example 2

```
void *consumer(void *consumer_thread_data) {
    ...
    while (!done()) {
        extracted = 0;
        while (extracted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 1) {
                extract_from_queue(&my_task);
                task_available = 0;
                extracted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
        process_task(my_task);
    }
}
```

Overheads of Locking

- Locks represent serialization points since critical sections must be executed by threads one after another.
- Encapsulating large segments of the program within locks can lead to significant performance degradation.
- It is often possible to reduce the idling overhead associated with locks using pthread_mutex_trylock.

Alleviating Locking Overhead

```
/*  using pthread_mutex_trylock routine */
…
Pthread_mutax_t tryLock_lock = PTHREAD_MUTEX_INITIALIZER;
…
lock_status=pthread_mutex_trylock(&tryLock_lock);
if (lock_status == EBUSY) {
    /* do something else */
    …
}
else {
    /* do one thing */
    …
    pthread_mutex_unlock(&tryLock_lock);
}
…
```

Condition Variables

- The third class of functions address communications between threads that share a mutex.
- A condition variable allows a thread to block itself until specified data reaches a predefined state.
- A condition variable indicates an event and has no value.
  - One cannot store a value into nor retrieve a value from a condition variable.
  - If a thread must wait for an event to occur, that thread waits on the corresponding condition variable.
  - A condition variable has a queue for those threads that are waiting the corresponding event to occur to wait on.
  - If another thread causes the event to occur, that thread simply signals the corresponding condition variable.

Condition Variables

- This class includes functions to create, destroy, wait and signal based upon specified variable values.
- Functions to set/query condition variable attributes are also included.
- A condition variable is always used in conjunction with a mutex lock.

Creating & Destroying Condition Variables

- Condition variables must be declared with type pthread_cond_t, and must be initialized before they can be used.
- There are two ways to initialize a condition variable:
  - Statically, when it is declared. For example: pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
  - Dynamically, with the pthread_cond_init() routine. This method permits setting condition variable object attributes, *attr* (which may be specified as NULL to accept defaults).
- **Routines:**

  pthread_cond_init (*condition, attr*)

  pthread_cond_destroy (*condition*)

  pthread_condattr_init (*attr*)

  pthread_condattr_destroy (*attr*)

Waiting and Signaling on Condition Variables

- **Routines:**
  pthread_cond_wait (*condition, mutex*)
  pthread_cond_signal (*condition*)
  pthread_cond_broadcast (*condition*)

- pthread_cond_wait() blocks the calling thread until the specified *condition* is signalled.
- This routine should be called while *mutex* is locked, and it will automatically release the *mutex* while it waits.
- After signal is received and thread is awakened, *mutex* will be automatically locked for use by the thread.
- The programmer is then responsible for unlocking *mutex* when the thread is finished with it.

Waiting and Signaling on Condition Variables

- pthread_cond_signal() is used to signal (or wake up) another thread which is waiting on the condition variable.
  - It should be called after *mutex* is locked, and
  - must unlock *mutex* in order for pthread_cond_wait() routine to complete.

- The pthread_cond_broadcast() routine unlocks all of the threads blocked on the condition variable.

Waiting and Signaling on Condition Variables

- Proper locking and unlocking of the associated *mutex* variable is essential when using these routines. For example:
  - Failing to lock the *mutex* before calling pthread_cond_wait() may cause it NOT to block.
  - Failing to unlock the *mutex* after calling pthread_cond_signal() may not allow a matching pthread_cond_wait() routine to complete (it will remain blocked).

Producer-Consumer Using Condition Variables

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
/* other data structures here */
...
main() {
    /* declarations and initializations */
    task_available = 0;
    pthread_cond_init(&cond_queue_empty, NULL);
    pthread_cond_init(&cond_queue_full, NULL);
    pthread_mutex_init(&task_queue_cond_lock, NULL);
    /* create and join producer and consumer threads */
    ...
}
```

Producer-Consumer Using Condition Variables

```
void *producer(void *producer_thread_data) {
    while (!done()) {
        create_task();
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 1)
            pthread_cond_wait(&cond_queue_empty,
                &task_queue_cond_lock);
        insert_into_queue();
        task_available = 1;
        pthread_cond_signal(&cond_queue_full);
        pthread_mutex_unlock(&task_queue_cond_lock);
    }
}
```

Producer-Consumer Using Condition Variables

```
void *consumer(void *consumer_thread_data) {
    while (!done()) {
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 0)
            pthread_cond_wait(&cond_queue_full,
                &task_queue_cond_lock);
        my_task = extract_from_queue();
        task_available = 0;
        pthread_cond_signal(&cond_queue_empty);
        pthread_mutex_unlock(&task_queue_cond_lock);
        process_task(my_task);
    }
}
```