STEM Education Sydney

# COMP2017

## Systems Programming

Computer Exam Review

Strategic Partners:

Pursue Education Alliance

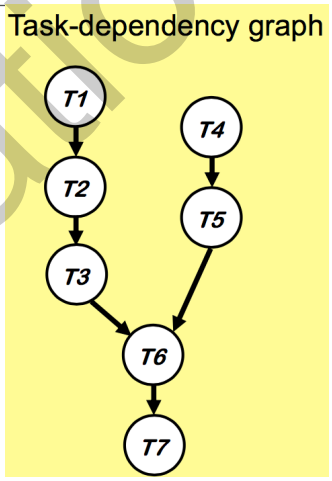
# COMP2017 Computer Exam Thread Revision

**Outline**

- Parallelism and Threads
  - Parallelism
  - POSIX Threads
- Lock-based Synchronisation
  - Race condition and critical section
  - **Mutex**
  - Deadlock
  - **Semaphore**
  - Lock Contention and Granularity, other issues with locks

## Parallelism and Threads

### Parallelism




For example, preparing a salad consists of 7 tasks(T1-T7).
A single chef prepares a salad in processing the 7 tasks sequentially (one after the other).
If we bring another chef that works in parallel, Task T1 is processed in parallel to Task T4. Processing tasks in parallel is called **task-parallelism**.

However, we can have several Chefs work in parallel on the "data" of a task. This form of parallelism is called **data-parallelism**.
Example: Chef 1 and Chef 2 chop one half of the lettuce each (Tasks T3a and T3b).

**Task-parallelism**: parallelism achieved from executing different tasks at the same time.
**Data-parallelism**: performing the same task to different data-items at the same time.

**Dependencies**: An execution order between two tasks Ta and Tb.
$T_a \rightarrow T_b$ Ta must complete before Tb can execute. Dependencies limit the amount of parallelism in an application.

### POSIX Threads

A **thread** is a series of instructions that are executed in the memory context of a process.
A thread shares the virtual memory of a process with all other threads.
- Shares heap, static/global, and code, env space

**Pthreads** is a library specified by the POSIX standard, defining an API for creating and manipulating threads. Most operating systems support **Pthreads**.

**Using Pthreads Library**

1. Include the <pthread.h> header in your source code.
2. Specify -lpthread flag to tell gcc to link in the POSIX thread library.

```
gcc -o hello hello.c -lpthread
```

**Create a thread: pthread_create()**

```
int pthread_create(pthread_t *tid, const pthread_attr_t *atrr, void* (*start_routine)
(void*), void* arg);
```

- *tid, a pointer to pthread_t used to store thread ID.
  - We need a way to refer to the thread, e.g. when we want to join it

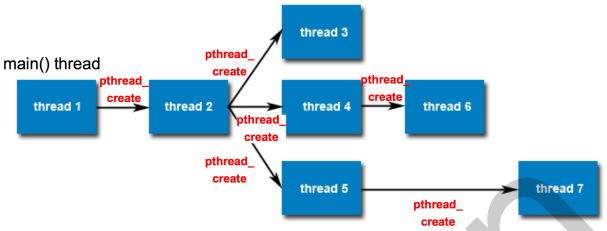
- *atrr, usually NULL, specifies default attributes.
- void* (*start_routine) (void*), a function pointer which points to a function that thread will execute. The function must return a void*, and take a void* as argument.
- *arg, the argument to be passed into the thread function at run-time.

Return 0 if success. If error occurs, an error number will be returned to indicate the error.

**Relationship between threads**

Threads are peers.

- A thread may create other threads.
- There's no implied hierarchy or dependency between threads.
- No assumption on execution order of siblings possible. The siblings may go first.




For Example: Assume thread2 creates thread3 and then thread4. You must not assume that thread3 will start execution before thread 4. Except when applying Pthread scheduling mechanisms.

Note: Main thread is a bit different. If we hit the return 0 in main, all other thread stops.

**Passing Arguments to threads**

Idea: Pass a pointer to the thread function.

Problems? If pass &i to the thread?

If we directly pass the &i to the thread, the index variable is shared between all threads.

The threads may not access the parameter fast enough to read the value “assigned” to them.

So this leads to disaster quickly.

(print_array_wrong.c)

For example, by the time thread0 dereferences the pointer to variable i, the value has already been changed by the main thread. Thread0 reads the wrong value (2 instead of 0).

```
1  #include<stdio.h>
2  #include<pthread.h>

3  #define NUMT ...
4  int args[NUMT];                /* argument array */
5  pthread_t threadIDs[NUMT];     /* thread IDs */

6  void * tfunc (void * p) {
7     printf("thread arg is %d\n", * (int *) p );
8  }

9  int main(void) {
10   int i;
11   for (i=0; i < NUMT; i++) {
12     args[i] = i; /* init arg for thread #i */
13     pthread_create(&threadIDs[i], NULL,
14                    tfunc, (void *) &args[i] );
15   }
16   for (i=0; i < NUMT; i++) {
17     pthread_join(threadIDs[i], NULL);
18   }
19   return 0;
20 }
```

Replace it with &i?

**Passing more than one argument to a thread**

**Idea**: Use a struct and pass the pointer to the struct as an argument. (pthread_more_arguments.c)

**Wait for a thread to terminate execution: pthread_join()**

int pthread_join(pthread_t tid, void** return_value);

- tid, The thread ID of the thread we are waiting for.
- **return_value, A pointer to the address of the return value of the joined thread. Usually stores the completion status of the exiting thread. If the thread wants to return multiple values, we return a pointer to a struct.

  If return_value is NULL, the return value of the thread(completion status) is ignored.

Return 0 if successful.

Note: Once a thread is joined, the thread no longer exists. Its thread ID is no longer valid, and it cannot be joined again.

**Terminate a thread**

- A thread can return from his start routine.
- A thread can call pthread_exit(NULL)
- A thread can be cancelled by another thread: pthread_kill().

In each case, the thread is destroyed and his resources become unavailable.

Note: Terminating a thread does not release dynamic memory allocated by the thread (malloc), as well as to close files that have been opened by the thread.

**Cancel thread by parent via signal: pthread_kill();**

```
int pthread_kill(pthread_t thread, int sig);
```

- thread, the thread id to sent signal to.
- sig, the signal to be sent to the thread.

# Lock-based Synchronisation

## Race Condition and Critical section

**Race Condition** is a situation in which multiple threads read and write a shared data item and the final result depends on the relative timing of their execution.
A race condition happens when two threads access a variable simultaneously, and (at least) one access is a **write**. Simultaneous reads of a variable are not a problem!

**Critical section**: Two or more *code*-parts that access and manipulate shared data(a shared resource).
Only 1 thread can execute within a critical section. We call this **mutual exclusion** between threads.

## Mutex

A **mutex** protects shared data by allowing only 1 thread into critical section.
Threads share the address space. Variables declared in main() before creating the thread is visible by both threads.

**Initialise mutex: pthread_mutex_t_init(), or PTHREAD_MUTEX_INITIALIZER**
When we initialise the mutex, it is **free**(or unlocked).

```
int pthread_mutex_t_init(
        pthread_mutex_t *restrict mutex,
        const  pthread_mutexattr_t *restrict attr
);
```

- mutex, a pointer to a pthread_mutex_t struct which should be declared beforehand.
- attr, specifies the attribute we need for setting up a mutex, usually NULL for the default attributes.

Return 0 if successful.

**Mutex Initialisation Styles**(mutex_styles.c)
**Style 1: Use pthread_mutex_init()**

```
pthread_mutex_t lock;      //Declare in either static space(static based) or in function at run-time(stack based)
pthread_mutex_init(&lock, NULL);        //Use the declared variable to initialise mutex at run-time
```

**Style 2: Use Macro**

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;        //Initialise the mutex, can be both in a function at run time(stack based) or outside the function(static based)
```

**Style 3: Dynamically allocate mutex on heap**

```
pthread_mutex_t *heap_mutex = (pthread_mutex_t *)malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(heap_mutex, NULL);        //Initialise the heap_mutex
```

Note: Attempting to initialize an already initialized mutex results in undefined behavior.

**Acquire a lock: pthread_mutex_lock()**

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

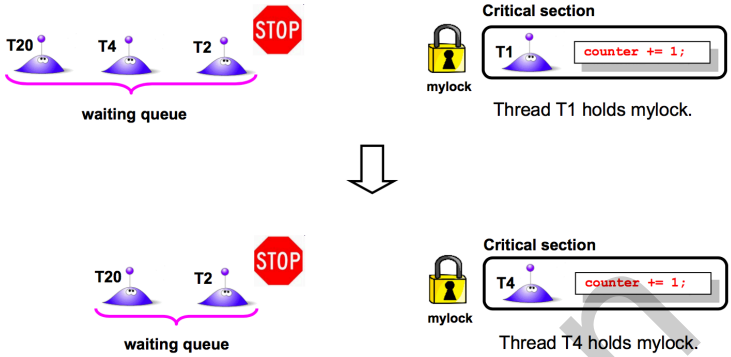
**pthread_mutex_lock()**

- If the lock is free, then calling thread will acquire it. Return 0 immediately.
- If the lock is held by other threads, the thread will be blocked until the lock is available.

**pthread_mutex_trylock()** is similar to lock, however,

- If the lock is not free, then it will return immediately. No blocking, no queuing.

Return 0, if we acquire the lock. Return non-zero if we don't get the lock.




Note: The pthread library does not guarantee fairness. Not necessarily the thread being blocked for the longest time will be selected.

**Release the lock: pthread_mutex_unlock()**

pthread_mutex_unlock(pthread_mutex_t *mutex);

Note: Always remember to release the lock when we finish with the shared resource.

**Destroy a mutex(to be uninitialized): pthread_mutex_destroy()**

int pthread_mutex_destroy(pthread_mutex_t *mutex);

---

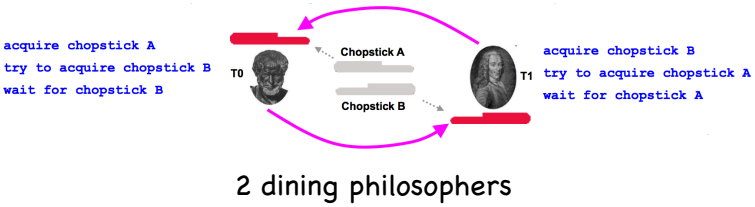
## Deadlock

A **deadlock** is a condition involving one or more threads and one or more resources (e.g. mutexes), such that each thread is waiting for one of the resources, but all the resources are already locks by separate threads, thus no more progress possible.

**Self-Deadlock**: A thread attempts to acquire a mutex that it already holds.

```
pthread_mutex_lock(&mylock); // Acquire mylock
pthread_mutex_lock(&mylock); // Try to acquire mylock again!
                             // Get blocked and wait for mylock
                             // to become available...
```

**ABBA-Deadlock**: 2 threads attempting to acquire 2 mutexes A and B. One in the order $A \rightarrow B$, the other in the order $B \rightarrow A$.



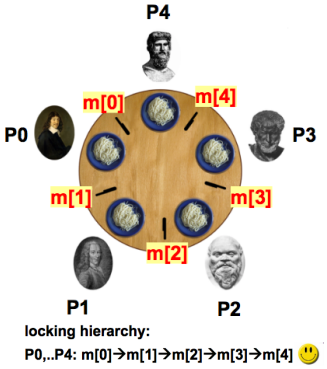

**Deadlock Prevention**

Prevent circular waits (cycles) by a **locking hierarchy**.

1. Impose an ordering on mutex acquisition.
   Example: $A \rightarrow B \rightarrow C \rightarrow D$
2. Require that all threads acquire mutexes in the same order.

Locking hierarchy requires programmers to know in advance what mutexes a thread will need.

Note: The order of unlock operations is immaterial!




Dining Philosopher Example(philosophers.c)

5 philosophers are sitting around a table to eat lunch. However there're only 5 chopsticks available. As long as one philosopher can pick up the chopsticks on the left hand side and right hand side, he can start to eat. After finishing eating, chopsticks are placed back to the same location. We treat philosopher as **threads**, and chopstick as resources.

```
#define MAX 5
pthread_t thr[MAX];
pthread_mutex_t m[MAX];

void * tfunc (void * arg) {
  long i = (long) arg; // thread id: 0..4
  for (;;) {
    pthread_mutex_lock( &m[i] );
    pthread_mutex_lock( &m[(i + 1) % MAX] );
    printf("Philosopher %d is eating...\n", i);
    pthread_mutex_unlock(&m[i]);
    pthread_mutex_unlock(&m[(i + 1) % MAX]);
  }
}
```

Deadlock possible

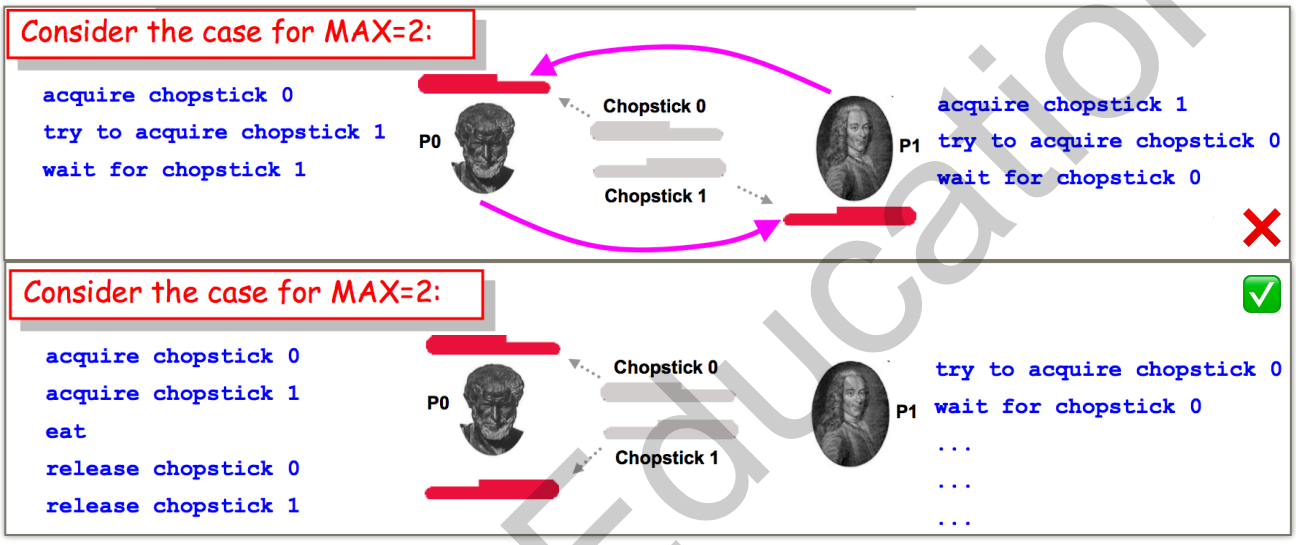
last philosopher does something different

Solution: Introduce a locking hierarchy:

1) Pick up the chopstick with the smaller index.
2) Pick up the chopstick with the higher index.

```
for (;;) {
    if ( i < ((i + 1) % MAX) ) {
      pthread_mutex_lock(&mtx[i]);
      pthread_mutex_lock(&mtx[(i + 1) % MAX]);
    } else {
      pthread_mutex_lock(&mtx[(i + 1) % MAX]);
      pthread_mutex_lock(&mtx[i]);
    }
    printf("Philosopher %d is eating...\n", i);
    pthread_mutex_unlock(&mtx[i]);
    pthread_mutex_unlock(&mtx[(i + 1) % MAX]);
}
```

Solution






## Semaphore <semaphore.h>

**Semaphores** are non-negative integer synchronization variables. It's similar to mutex but we use integer $s$ to represent the lock.

**2 basic operations on semaphores**

- $P(s)$: Wait and get the lock: A thread has to wait until the value of $s$ is greater than 0. If it is true, then $s$ is decremented by 1, and the thread is allowed to continue execution.

```
while(s==0) wait();        //Wait until s>0, i.e. lock is available for us to get
s--;                       //Get the lock
```

- $V(s)$: Release the lock.

```
s++;
```

- At any time, only one $P(s)$ or $V(s)$ operation can modify $s$.
- At any time, $s \geq 0$

**Initialise Semaphore: sem_init()**

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- *sem, a pointer to sem_t struct to initialise.
- pshared, usually 0 to indicate this semaphore is shared between threads.
- value, initialise $s$. If 0, then the lock is in a locked state at beginning. Other non-zero value, for example 1, means that only 1 thread can use the shared resource.

If value=1, we call it **binary semaphore**, which achieves mutual exclusion between threads. And behaves like a mutex.

Otherwise, we call it **counting semaphore**, which allows $N$ threads at the same time in the critical section.

```c
#include <semaphore.h>

#define MAX 4
#define MAX_ITER 50000000
pthread_t thr[MAX];
sem_t s;

long counter = 0;

void * tfunc (void * arg) {
  int i;
  for (i=0; i<MAX_ITER; i++) {
    sem_wait(&s);
    counter++; //critical sectior
    sem_post(&s);
  }
}

int main() {
  int i, j;
  sem_init(&s, 0, 1);
  ...
}
```

semaphore.c

**Wait and obtain the lock(decrement semaphore): sem_wait()**

int sem_wait(sem_t *sem);

sem_wait() decrements (locks) the semaphore pointed to by sem.

- If the semaphore's value >0 , then the decrement proceeds, and the function returns immediately.
- If the semaphore currently==0, then the call blocks until it becomes possible to perform the decrement (i.e., the semaphore value rises above zero).

**Release the lock(increment semaphore): sem_post()**

int sem_post(sem_t *sem);

If the semaphore's value consequently becomes >0, then another process or thread blocked in a sem_wait() call will be woken up and proceed to lock the semaphore.

**Destroy semaphore(to be uninitialised): sem_destroy()**

int sem_destroy(sem_t *sem)

**Synchronizing Threads with Semaphores**

- Assume there're 2 threads. $T_1$ executes function T1, $T_2$ executes function T2.
- We set up a semaphore with initially locked: sem_init(&s, 0, 0)
- $T_1$ has a sem_post() operation
- $T_2$ has a sem_wait() operation.
- When T2 reaches sem_wait(), it will block until T1 has executed sem_post().

```c
sem_t s;

void * T1(void * arg) {
  ...
  printf("this comes first\n");
  sem_post(&s);
  ...
}

void * T2(void * arg) {
  ...
  sem_wait(&s); //wait for T1
  printf("this comes second\n");
  ...
}

int main() {
  sem_init(&s, 0, 0);
  ...
}
```

Have to wait until T1 posts

Example(pingpong.c): Assume 2 threads, executing the thread routines T1 and T2, respectively.
Thread T1 outputs “ping” in an endless loop.
Thread T2 outputs “pong” in an endless loop.

**Comparison with Semaphores and Mutexes**

**Mutex≈Binary Semaphore**, which allows only 1 thread at a time to execute a critical section.

1. A mutex that is locked has an **owner**, only the thread acquires a mutex should release it!
2. A mutex is initially unlocked.
   Semaphores can be initialized to any value, including 0. A semaphore initialized to 0=locked
3. A mutex can only assume two states, “unlocked” and “locked”.
   A semaphore is a counter. Counter values are in the range 0 .. N.



## Lock Contention and Granularity, other issues with locks

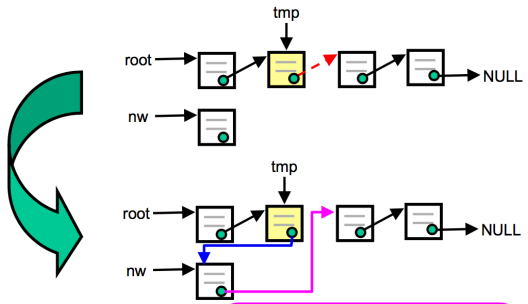
**Lock Contention**: A highly contended lock has many threads waiting to acquire it.

- Lock frequently obtained, or held for a long time, or both.
- All threads queue up, trying to acquire the contended lock.

**Example: Insert a new node to a linked list**

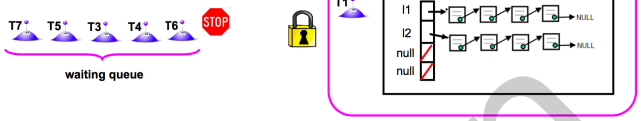
- Create a new node nw
- Create a temporary pointer tmp to the node after which the new node is to be inserted.
- Update pointers: set nw.next = tmp.next, and tmp.next = nw.

Now assume now that two threads attempt to insert a new node into the list simultaneously.
Both attempt to insert the node after the yellow node. It's a race condition.
If multiple threads use our linked list, we need to synchronize access to lists.




**Solution 1: Coarse-grained lock**—Use one mutex to protect all the lists.

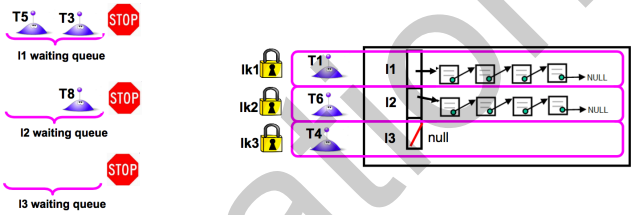
- Now all threads have to block at a single mutex to wait for list access. Even if they want to access different lists!




**Solution 2: Medium-grained lock**—Use one mutex for each linked list.

- Now the threads have to block at the list that they want to access.
- Access to different lists can happen simultaneously.

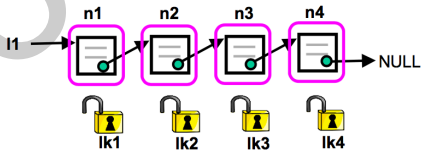
Problem: What if a thread wants to move a list node from list l1 to list l2?




**Solution 3: Fine-grained lock**—Use one mutex for each list node.

- Now threads can lock individual list nodes.
- Access to different list nodes can happen simultaneously.

Note: For some operations, it might be necessary to lock more than one node! So it's important to obey locking hierarchy to prevent deadlock!




**Locking granularity** describes the amount of data that a lock protects.

- A **coarse-grained lock** protects a large amount of data.
- A **fine-grained lock** protects only a small amount of data. Best performance, but hard to design.

Coarse-grained locks are likely to be highly contended.

**Other issues with locks**

1. **Locks** are **advisory** and **voluntary**. Compiler won't notify you when you have one critical section in your code unsurrounded.
2. **Livelocks**: Two or more threads are busy synchronizing and do not make progress in what they actually want to compute.
3. **Starvation**: One thread is never allowed into the critical section.

We need some fairness-property to prevent starvation and ensure that every thread is able to make progress.

**Lock and Lock-based Synchronization Summary**

Mutexes, semaphores are all **locks**, which have one commonality: They protect shared data such that only one thread can be in the critical section at any time. Other threads attempting to enter the critical section have to wait (block).
They protect shared data by locking a **critical section** against multiple entry of threads.
Synchronization via locks is called **lock-based synchronization**.

**Practical Exam: Processes(Week7-8) and Threads(Week9-10).**
**Be familiar with all functions mentioned in the summary!**