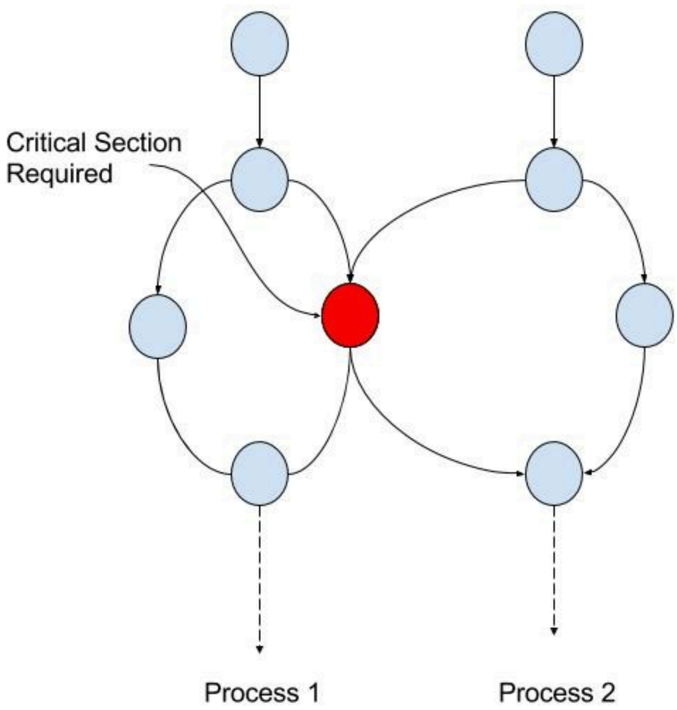
Week 11

Start at 4:05

P1: Interleavings of Threads

Critical section:Two or more code-parts that access and manipulate shared data (aka a shared resource).

Race condition: “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”.



P2: Lock-based Thread Synchronisation

Mutex

> Example

Blocking

```
pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;

void * thread_function(void * arg) {
    /*
      Assume There are 3 threads.

      T1 locks and executing critial section
      T2, T3 are waiting in the line `pthread_mutex_lock(&mylock);`

      Do not know after unlock mutex which thread will execute (order)
    */

    pthread_mutex_lock(&mylock);
    counter = counter + 1; //critical section
    pthread_mutex_unlock(&mylock);
}
```

Non-blocking

```
pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;

void * thread_function(void * arg) {
    /*
      Assume There are 3 threads.

      T1 try lock successfully and this function locks automatically
and executing critial section
      T2, T3 try lock fail and printf("Unscu...."), continue executing
    */
    if ( 0 != pthread_mutex_trylock(&mylock) ) {
      printf("Unsuccessful attempt to acquire lock\n");
```

```
      // #######  pthread_mutex_unlock(&mylock); // Error: mutex not acquired!  ==> Only the thread that owns a mutex should unlock it! #########
      // what happens => unlock thread2


    } else {
        // critical section start:
        //...      ==>  thread1 still goes here => can cause race condition as well
        // crtical section end
        pthread_mutex_unlock(&mylock);
    }
}
```

Dynamic creation of mutexes

```
pthread_mutex_t * mylock;

mylock = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(mylock, NULL);

...

pthread_mutex_destroy(mylock);
free (mylock);
```

Serialization

```
/*
  Part 1, Serialization: one thread has to wait another
*/
unsigned long counter=0;

```

```c
void * thread_function(void * arg) {
  long l;
  for (l=0; l < MAX_ITER; l++) {
    pthread_mutex_lock(&mylock);
    counter = counter + LongComputation();

    pthread_mutex_unlock(&mylock);
  }
}




/*
  Part 2, Reduce Serialization
*/
unsigned long counter=0;

void * thread_function(void * arg) {
  long l, tmp;
  for (l=0; l < MAX_ITER; l++) {
    tmp = LongComputation();

    pthread_mutex_lock(&mylock);
    counter=counter+tmp;//crit.sect.
    pthread_mutex_unlock(&mylock);
  }
}
```

P3: DeadLock

Example

```
T1                              T2



cat 1                           fox 1
dog 2                           dog 3 (blocking)
fox (blocking)



Can not reach next,  can not continue to unlock

free...                         free...
```

Necessary conditions for a deadlock

1. Mutual exclusion: a resource can be assigned to at most one thread.
2. Hold and wait: threads both hold resources and request other resources.
3. No preemption: a resource can only be released by the thread that holds it.
4. Circular wait: a cycle exists in which each thread waits for a resource that is assigned to another thread.

Using 10 minutes to write Q2, back at 4:50

DO not refer to Lecture answer, write by yourself

Deadlock Prevention

Cycles can prevented by a locking hierarchy: ==> breaking rule 4

1. Impose an ordering on mutexes.
2. Require that all threads acquire mutexes in the same order.

===> a special case

Thread 0:

```
acquire mutex A
for(;;) // enter endless loop
        // never free mutex A
```

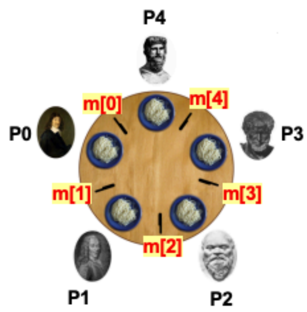
Thread 1:

```
try to acquire mutex A
wait (block) forever for mutex A
```

Example: Dining Philosopher's Deadlock (Skip this)

```
#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]);
  }
}
```

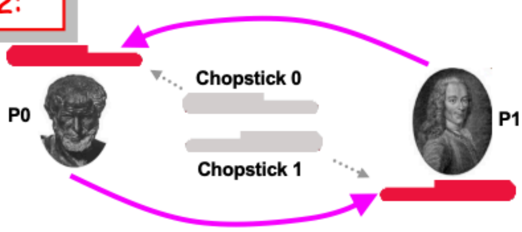


P0,..P3: m[0]→m[1]→m[2]→m[3]→m[4] 🙂

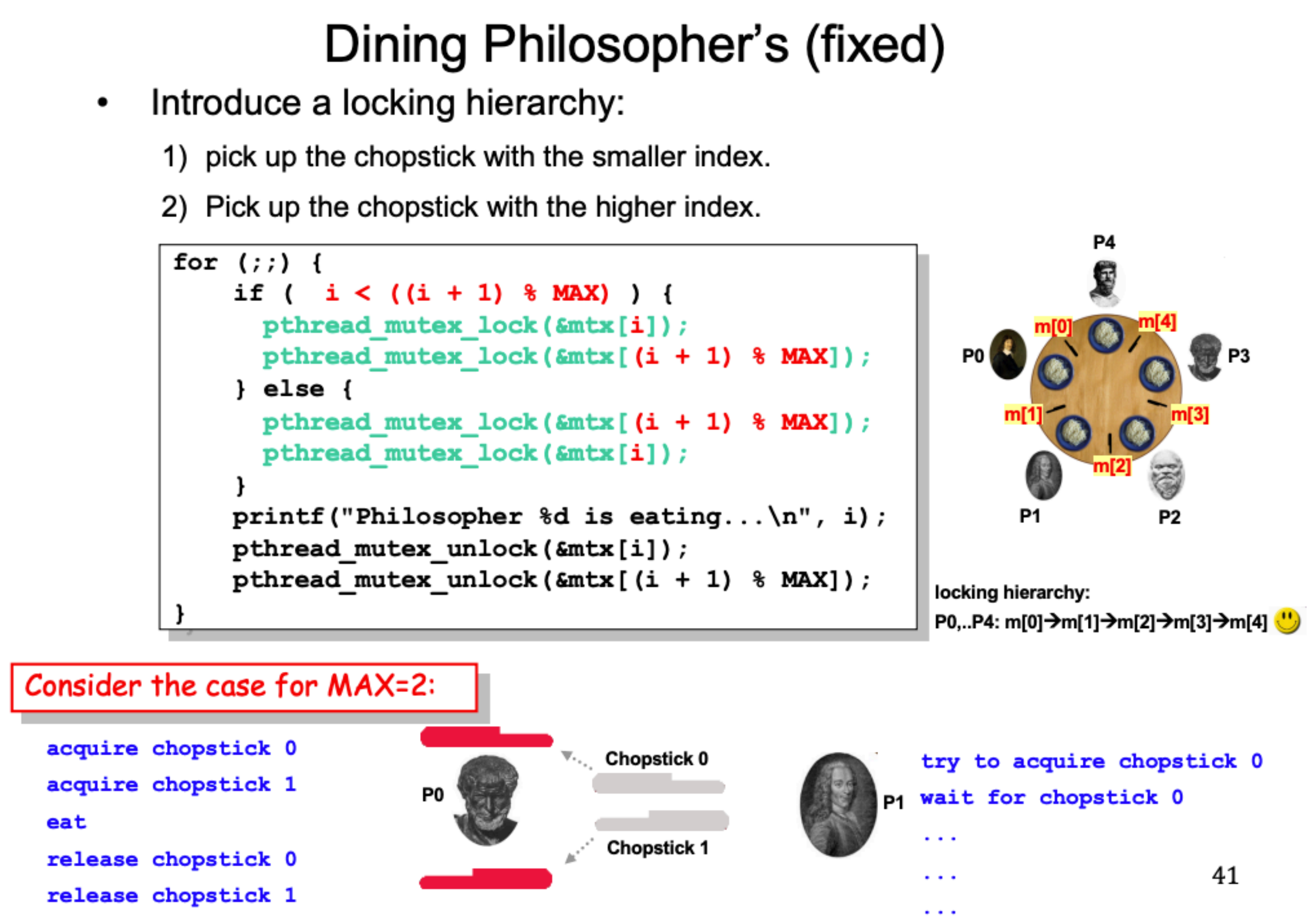
P4: m[4]→m[0] ☹

Consider the case for MAX=2:

```
acquire chopstick 0
try to acquire chopstick 1
wait for chopstick 1
```




```
acquire chopstick 1
try to acquire chopstick 0
wait for chopstick 0
```



Hold and wait ==> breaking rule 2

```
T1



cat
  operation(cat)
dog
  unlock cat
  lock(dog)
  operation(dog)
fox
  unlock dog
  lock(cat, fox)
  operation(cat, fox)
```

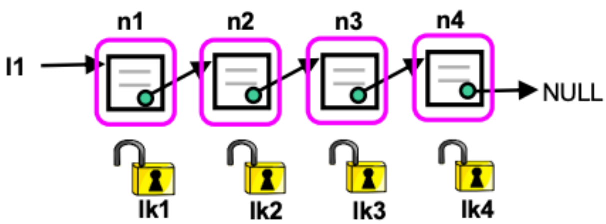
> Week11/Q1, Q2

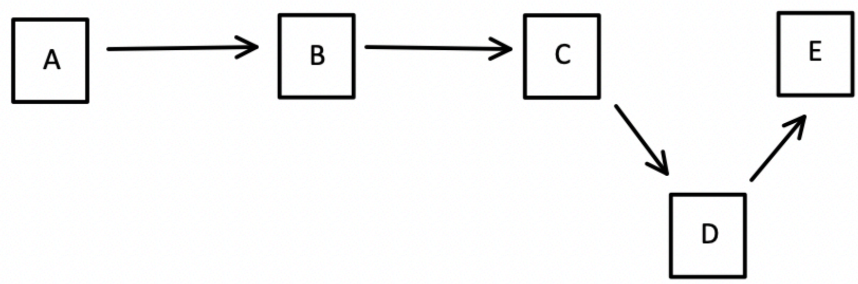
Starvation and Live-lock (Extension)

Starvation happens when “greedy” threads make shared resources unavailable for long periods. For instance, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

P4: Lock Contention and Scalability

- Coarse-grained lock => a lock for the whole linked list => no parallel => we can not do search in parallel
- Medium-grained lock.
- **Fine-grained lock.**






```
lock A ?
--- No

lock B ?
--- No

lock C ?
--- Yes       ==> the other thread can not access nodes after C
              ==> some race condition may happens when two threads
both inserting
                   after C without lock

lock D as well
update(C, D)

unlock C, D
```

P5: Semaphores

Semaphores are non-negative integer synchronization variables.

If you want to do something with an order.

Something like wake up using signal are without order

```
s has some init value maybe 0



V(s): [ s++; ]             ==> sem_post



P(s):
[
  while (s == 0) {
    wait();
  }
  s--;
]                           ==> sem_wait



The statements between brackets [ ] are therefore an atomic
operation.
==> At any time, only one P() or V() operation can modify s.
```

Example:

```
#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 section
    sem_post(&s);
```

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

Example: Dining Philosophers

```
sem_t chpstcks[N], limit;

void * Philosopher(void * arg) {
  long id = (long) arg;
  for(;;) {
    think();
    sem_wait(&limit);
    sem_wait(&chpstcks[id]);
    sem_wait(&chpstcks[(id+1)%N];
    eat();
    sem_post(&chpstcks[id]);
    sem_post(&chpstcks[(id+1)%N]);
    sem_post(&limit);
  }
}

int main() {
  int i;
  for(i=0; i<N; i++)
    sem_init(&chpstcks[i], 0, 1);
  sem_init(&limit, 0, N-1);
  ...
}
```

- A counting semaphore can prevent the Dining Philosophers from dead-locking:
  - Assume N philosophers sitting at the table.
  - Use a counting semaphore with an initial count of N-1.
    - → At most N-1 philosophers can pick up the left chopstick at once.
    - → At least one of those philosophers will have access to two chopsticks. This philosopher can eat.

Synchronizing Threads using Semaphores

```
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);
  ...
}
```

- Besides mutual exclusion, semaphores can also be used to synchronize threads.
- Example:
  - Assume 2 threads executing the thread routines T1 and T2.
  - Assume a semaphore s.
    - s is initiallized to 0.
  - T2 has a sem_wait() operation on s.
  - T1 has a sem_post() operation.
  - When T2 reaches sem_wait(), it will block until T1 has executed sem_post().
  - Question: what happens if T1 executes sem_post() before T2 executes sem_wait() ?

Question: what happens if T1 executes sem_post() before T2 executes sem_wait() ?

Synchronizing Threads using Semaphores

```
sem_t ________________;

void * T1(void * arg) {
  for (;;) {
    ____________________
    printf ("ping\n");
    ____________________
  }
}

void * T2(void * arg) {
  for (;;) {
    ____________________
    printf ("pong\n");
    ____________________
  }
}

int main() {
  ____________________
  ____________________
  ...
}
```

- Example:
  - 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.
  - How can we synchronize T1 and T2 using semaphores, such that the output will be

    ping
    pong
    ping
    pong
    ...

  ?

58

> Week11/Ping-Pong , Q3 & Q4
>
> Before Q4, Intro Barrier

10 minutes Break => continue Q3 back 5:23

```
We have previously solved the dining philosophers problem by using a
locking hierarchy.
This time, use a semaphore for the table that only allows N/2
philosophers to eat at a time
```

DO Question 4 ==>

Come up with an idea

10 min back at 5:50

P6: Amdahl’s Law

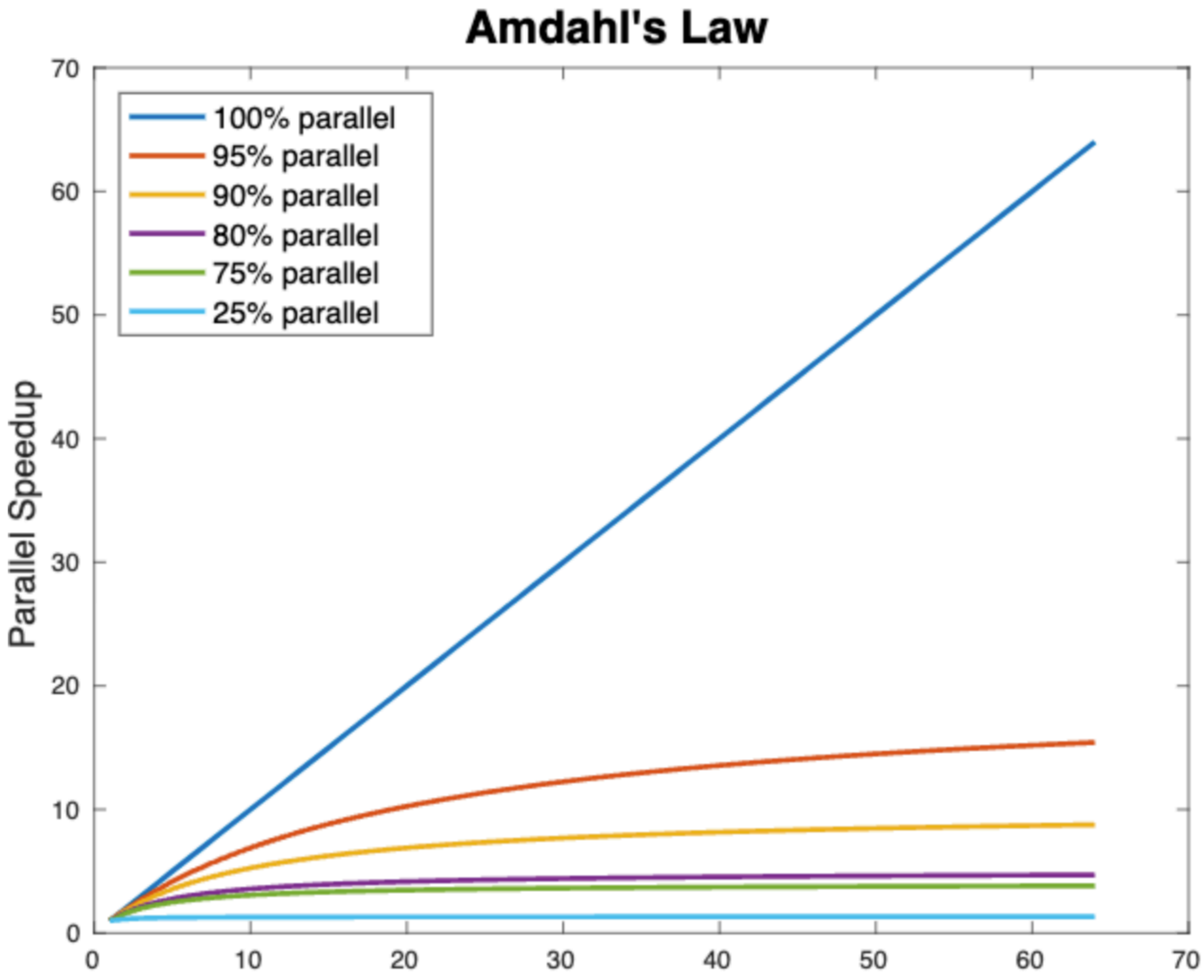
- p = fraction of work that can be parallelized.
- n = the number of threads executing in parallel.

$$Speedup = \frac{old\_running\_time}{new\_running\_time} = \frac{1}{(1 - p) + \frac{p}{n}}$$

(1 - p) => the part can not be parallelised

P => the part can be parallelised, and we have n workers.

If 100% => p = 1 => speed = n (linear)


Amdahl's Law
100% parallel
95% parallel
90% parallel
80% parallel
75% parallel
25% parallel
Parallel Speedup
0
10
20
30
40
50
60
70

> Week11/Q2