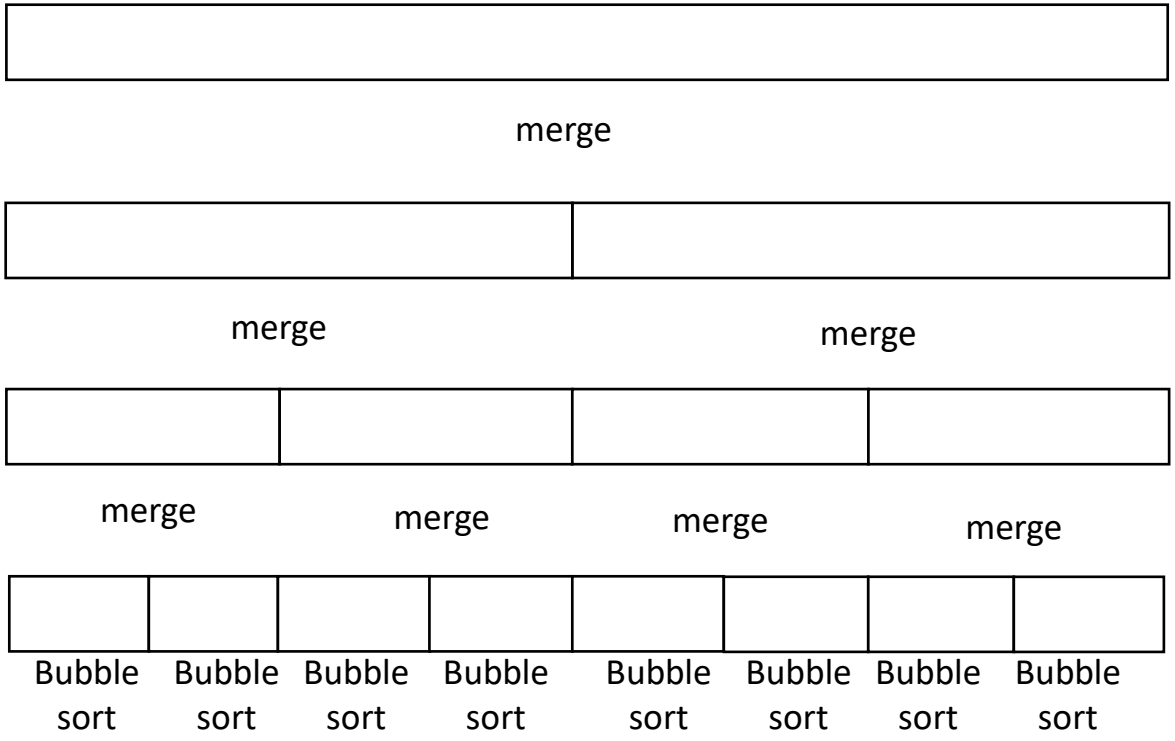
Operating Systems Programming Assignment #3

Parallel Merge Sort with Threads

Prof. Li-Pin Chang

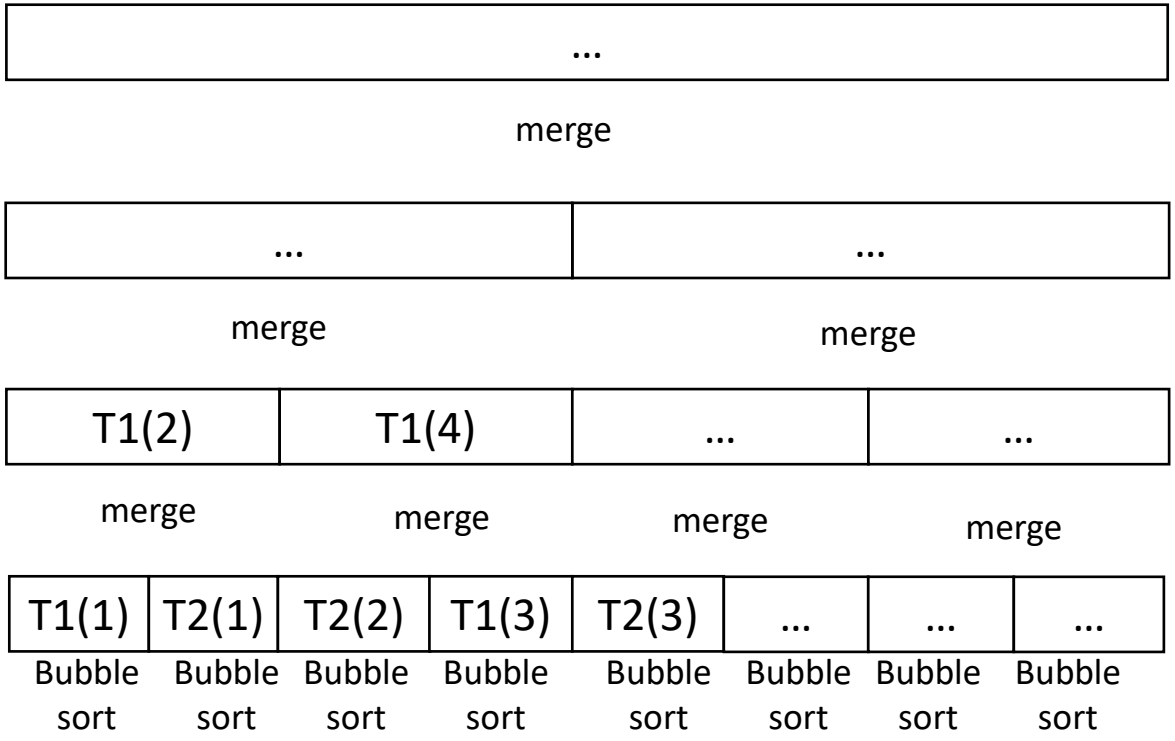
CS@NYCU

Jobs for Parallel Merge Sort



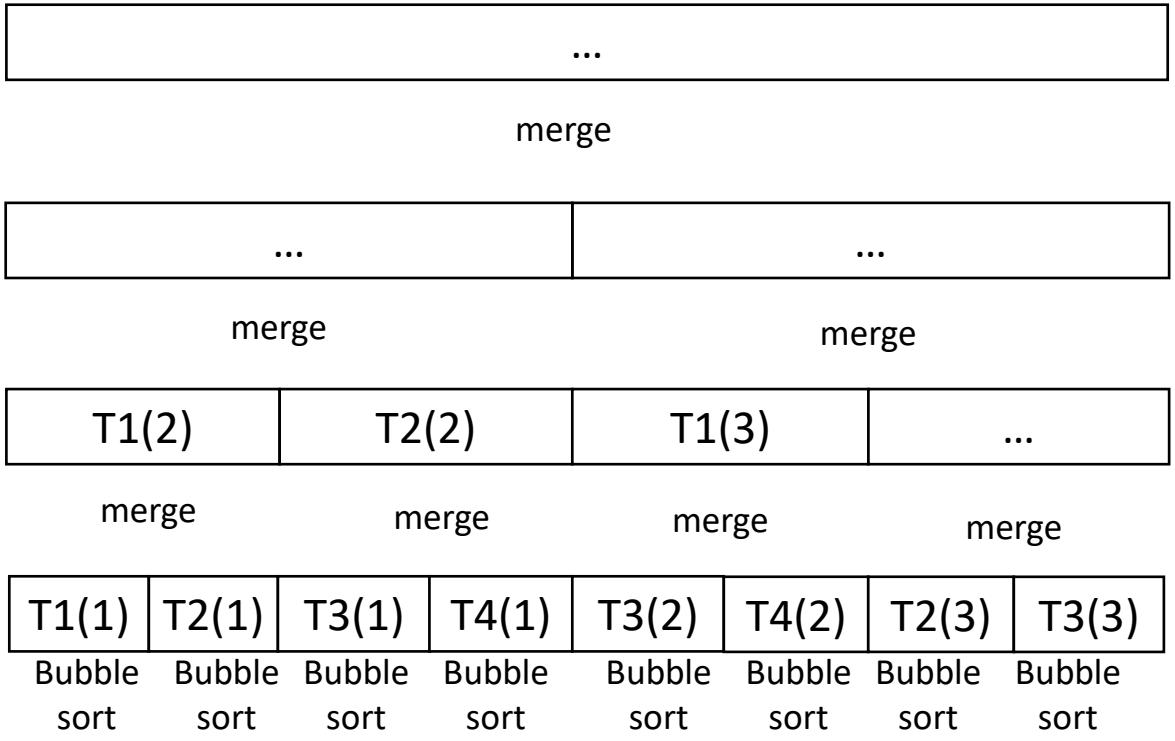
** In this assignment, the original array is divided into eight equal pieces for merge sort

Merge Sort, 2 Worker Threads



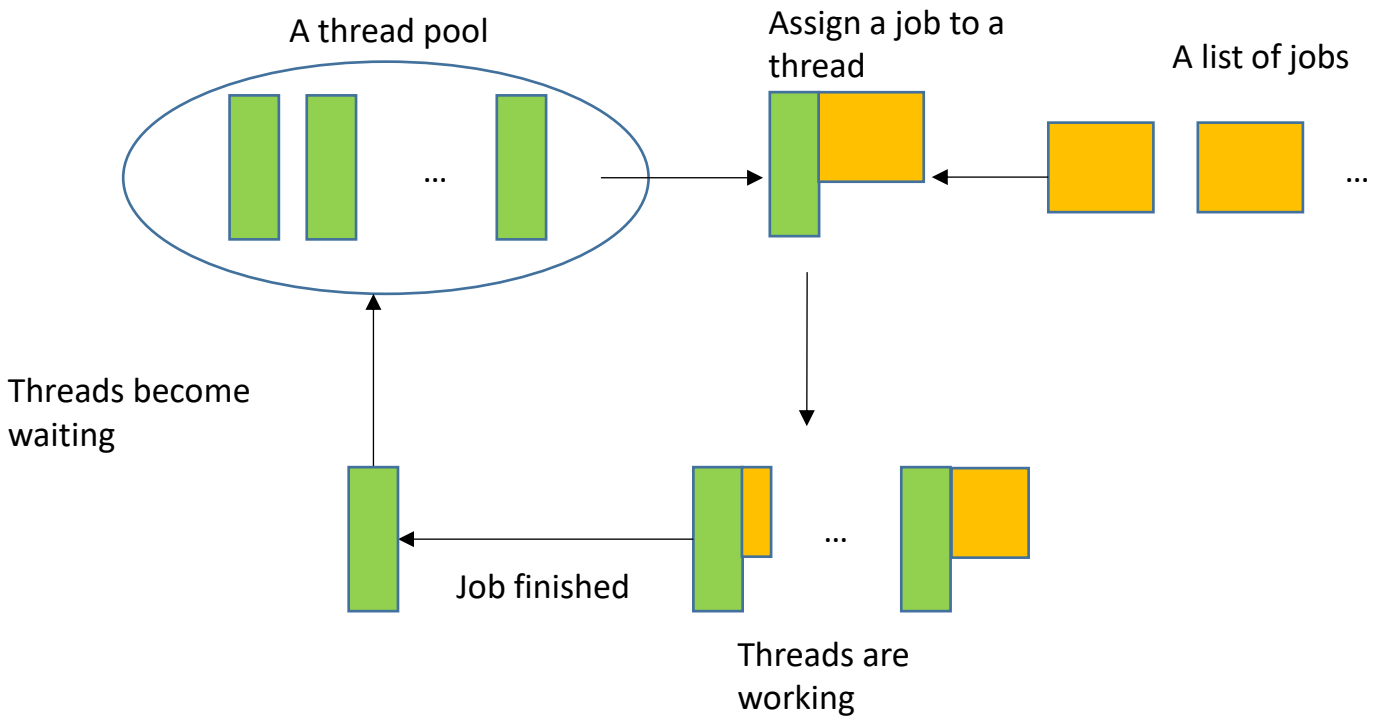
T1(3) = the 3rd invocation of worker thread T1

Merge Sort, 4 Worker Threads



T1(3) = the 3rd invocation of worker thread T1

The Concept of a Thread Pool



Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore S – integer variable
  - Initial value of S cannot be negative
  - Can only be accessed via these two indivisible (atomic) operations: wait() and signal()
  - An internal (invisible) waiting queue to manage blocked threads/processes

Synchronization with Semaphore

- Implementation of wait:

```
wait (S){
        S--;
        if (S < 0) {
                        add this process to waiting queue
                        block();  }
  }
```

- Implementation of signal:

```
Signal (S){
        S++;
         if (S <= 0) {
                        remove a process P from the waiting queue
                        wakeup(P);  }
}
```

Mutual exclusion
Semaphore mutex=1

Ti

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

Tj

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

Event
Semaphore synch = 0

Pi

```
S1;
signal(synch);
```

Pj

```
wait(synch);
S2;
```

Event
Semaphore synch = 0

Ti

Tj

Tk

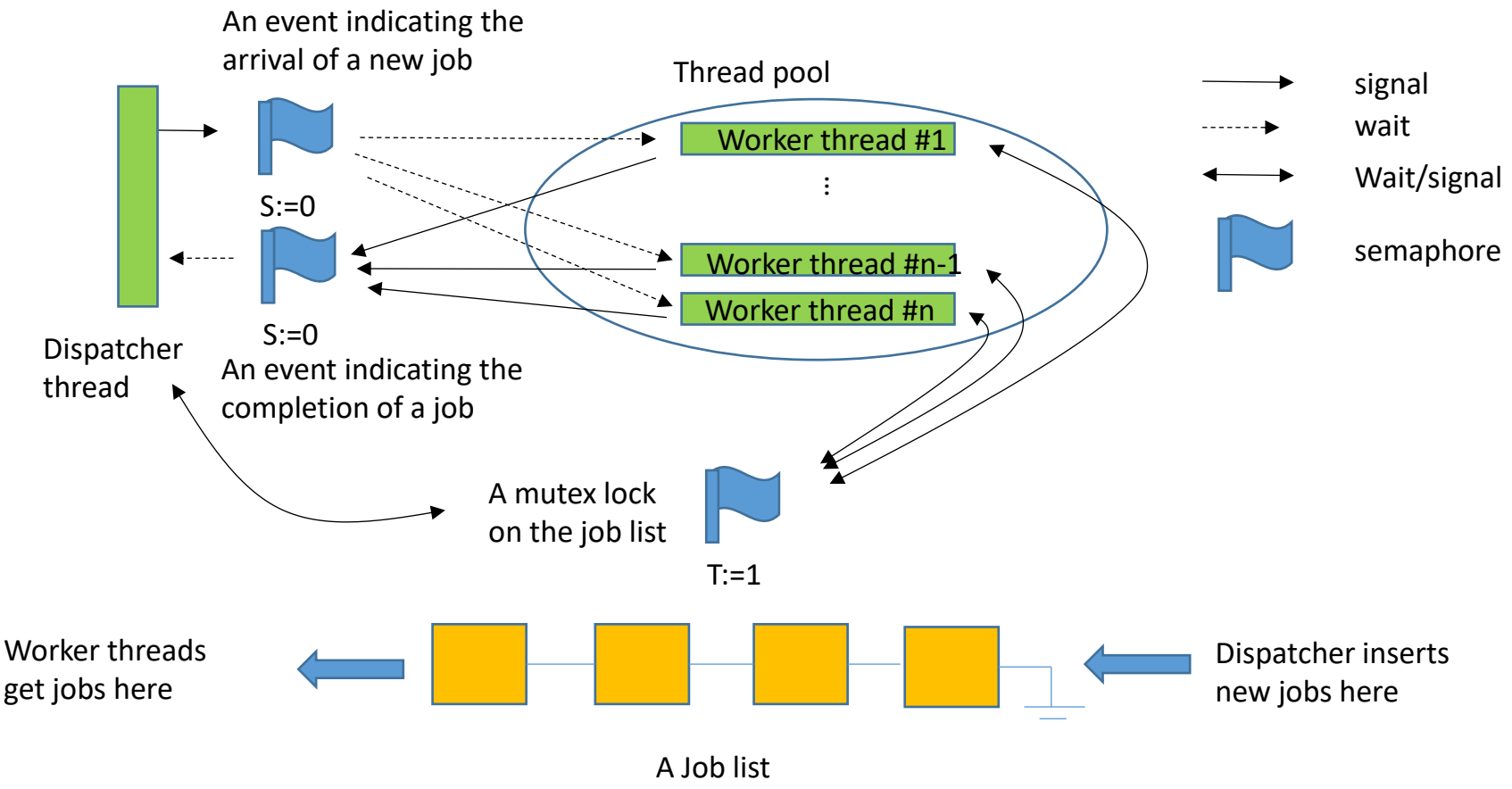
```
S1;
signal(synch);
```

```
wait(synch);
S2;
```

```
wait(synch);
S2;
```

Utilizing the internal waiting queue

A Reference Design



You don't have to implement this way, but your sorting results and execution time trend must be correct

A Reference Design

- Initially, the dispatcher thread inserts eight sort jobs for the eight bottom-level arrays and signals worker threads
- When a worker thread is signaled, it gets a job from the job list
- When a worker thread completes a job, it notifies the dispatcher
- When being notified, the dispatcher checks if any two pairing (buddy) sub-arrays have been sorted. If so, it inserts a new job of merging the two sub-arrays to the job list and signal a worker thread

Procedure

1. Read data from the input file “input.txt”
2. n=1
3. Do the sorting with a thread pool of n threads
4. Print the execution time
5. Write the sorted array to a file
   - Filename: output_n.txt (e.g., output_3.txt if n=3)
6. n++; if n<=8 then goto 3

Input Format

- Format of “input.txt”:

<# of elements of array><space>\n

<all elements separated by space>

  - Largest input: 1,000,000 integers
  - Generate your own file for testing

- Output file format “output_?.txt”:

<sorted array elements separated by space>

Output Format

- On the screen

```
worker thread #1, elapsed 19052.239000 ms
worker thread #2, elapsed 10029.638000 ms
worker thread #3, elapsed 7157.280000 ms
worker thread #4, elapsed 5263.625000 ms
worker thread #5, elapsed 5481.798000 ms
worker thread #6, elapsed 5884.612000 ms
worker thread #7, elapsed 5862.892000 ms
worker thread #8, elapsed 5037.831000 ms
```

- In the output files: sorted integers

APIs

- <pthread.h>

  Thread management

  - pthread_attr_init, pthread_create, pthread_exit

- <semaphore.h>

  Semaphore operations

  - sem_init, sem_wait, sem_post, sem_getvalue, sem_destroy

Remarks

- Numbers must be sorted in the ascending order
- Use bubble sort on the bottom-level arrays
- All the 8 output files must be identical
- Avoid polling in any place of your program
- Jobs must be dynamically created. Do not pre-generate all jobs
- You get 0 mark if you use quicksort() in any place
- Execution time decreases as *n* increases; Performance improv. saturates when *n* is large
  - Again! Why? Try googling “Amdahl's Law”

Testing OS Environment

- Ubuntu 22.04+
- Physical installation, VM, or WSL