Creational design patterns
Factory Method

• Lets a class defer instantiation to subclasses
  • No need to decide which subclass I want to use statically

• Example
  • Date (US style, Europe style, Chinese style, ...)
  • Window
Class diagram

Factory design pattern is somewhat similar with Strategy design pattern
When to use factory design pattern?

• The type of the sub-class is determined at run time
• The type changes very infrequently once set
Abstract Factory

• For creating families of related or dependent objects without specifying their concrete classes

• Examples
  • Date, currency, data
  • Window, mouse, scroll bar, ...

Abstract factory design pattern is somewhat similar with Visitor design pattern
Class diagram
We didn’t have time to cover any of the following content on Nov. 19th. They will be covered on Thursday Nov 21st lecture.

Parallel Programs
What is parallelization and why?
What is parallelization and why?

- **Conceptual reason**
  - Sometimes it gives you conceptual isolation among parallel units

- **Performance reason**
  - True parallelism: get multiple CPUs running at the same time
  - Concurrency: keep the CPU utilization high, while some concurrent units are waiting for I/Os

- **Modern hardware/system trend**
  - Multicore computers
  - Distributed systems
How to parallelize a sequential algorithm?

- Data parallelization
- Task parallelization
- Pipeline parallelization
Examples

• Matrix addition
  • Trivial data parallelism
  • Pay attention to row/column memory layout

• Array summation
  • Easy data parallelism, but we cannot follow the original sequential implementation where there is dependency among loop iterations
  • Cut the array to sub-arrays, get sub-array sum, aggregate

• Array sorting
  • Quicksort
  • Mergesort
  • Bubblesort
Principle

• We want parallel running code to have as little data/control dependence with each other as possible

• When there is dependence, synchronization is needed, which will cause slowdowns
  • If we forget to use synchronization, concurrency bugs (races) would happen
Matrix addition

• How to parallelize it?

• Use data parallelization
  • When we want to do the same operation repeatedly on many different variables/data, we can let the operation for different data conducted in parallel
  • Suppose we have K CPU cores, we can let each core work on N/K rows (N is the dimension of the matrix row)
Quicksort

```
algorithm quicksort(A, lo, hi) is
  if lo < hi then
    p := partition(A, lo, hi)
    quicksort(A, lo, p - 1)
    quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
  pivot := A[hi]
  i := lo - 1
  for j := lo to hi - 1 do
    if A[j] ≤ pivot then
      i := i + 1
      swap A[i] with A[j]
  swap A[i+1] with A[hi]
  return i + 1
```
How to parallelize quicksort?

• Run the two quicksort in parallel
Mergesort

Divide the unsorted list into $n$ sublists, each containing 1 element

Repeatedly merge sublists to produce new sorted sublists, until there is only 1 sublist remaining
How to parallelize merge-sort?

• Run the merge sort on different sub-lists in parallel

• Merge-sort is among the easiest to parallelize sorting algorithms
procedure bubbleSort( A : list of sortable items )
    n = length(A)
    repeat
        swapped = false
        for i = 1 to n-1 inclusive do
            /* if this pair is out of order */
            if A[i-1] > A[i] then
                /* swap them and remember something changed */
                swap( A[i-1], A[i] )
                swapped = true
            end if
        end for
    until not swapped
end procedure
Bubble sort

• Bubble sort is extremely difficult to parallelize because there are strong dependency among loop iterations
A more difficult example (pipeline parallelism)

while (! End of source file)
    read a line
    process the line
    write the processing result to destination file
How to parallelize?

Use pipeline parallelism: run three threads as following

CPU1: Read line 1 → Process line 1 → write result 1 → read line 4 → process line 4
CPU2: read line 2 → process line 2 → write result 2 → read line 5 → ...
CPU3: read line 3 → process line 3 → write result 3 → read line 6 → ...
Another example

• Computing the sum of an array

```c
sum = 0;
for ( i=0; i< M; i++)
    sum = sum + A[i];
printf ("sum is %d", sum);
```
Another example (Cont’d)

• Computing the sum of an array

```c
sum = 0;
for ( i=0; i< M; i++)
    sum = sum + A[i];
printf ("sum is %d", sum);

for (i =0 ; i<M/4; i++)
    sum1 = sum1 + A[i]
For (i=M/4; i<M/2; i++)
    sum2 = sum2 + A[i]
...
...
Sum1+sum2+sum3+sum4
```

Sometimes, we need to slightly transform the original sequential code to see parallelization opportunities
Summary: 3 ways to parallelize a sequential alg.

• Data parallelization
• Task parallelization
• Pipeline parallelization
Performance Analysis
How to estimate the benefit of parallelization?

• What is the ideal speedup on N machines/cores?
  • Nx

• Why cannot we achieve the ideal speedup?
  • Thread-related overhead
  • Non-CPU resource limitation
  • non-parallelizable code component
How to estimate the benefit of parallelization?

• What is the ideal speedup on N machines/cores?
  • Obviously N

• Why cannot we achieve the ideal speedup?
  • Algorithm reasons
  • Practical reasons
What affect real parallelization efficiency?

• Amdahl's law
  • [https://www.techopedia.com/definition/17035/amdahls-law](https://www.techopedia.com/definition/17035/amdahls-law)

• Critical path
  • You can represent a parallel program in a DAG, with an edge representing a task cannot start until another one finish
  • The longest path in your DAG is called critical path
  • The length of the critical path determines the execution time of your program with unlimited number of processors
How to estimate the benefit of parallelization?

• Amdahl's law
• Critical path

• Load imbalance
• Resource competition
• Data sharing cost (false sharing leads to huge performance lost)
• Synchronization overhead (lock, etc.)
• Other parallelization overhead (i.e., data duplication, work duplication, and aggregation)

Lecture ends here
We didn’t cover this section in lectures. You need to be able to read multi-threaded C code, but you are not required to write them in exam.

Parallelization
implementation

Threads, Processes
APIs
See my code examples
Principles

• What are shared data and what are private data
• When and how to do synchronization
Thread

• Thread creation
• Thread join
• Lock
  • Data race bugs
  • Deadlock bugs
• What are shared?
  • Global variables (and heap objects)
• What are private?
  • Stack variables
Data race bugs

• Threads competing on accessing shared variables
• Toy example

Thread 1                                Thread 2

tmp = x;
tmp = x;
x = tmp+1;
x = tmp+1;

//the result of x could be 1 or 2, assuming x was 0 at the beginning
How to avoid data races

• Using locks to enforce atomicity (also called mutual exclusion)
  Lock (l)
  \( X++; \)
  Unlock (l)

• Using signal/wait to enforce ordering
Deadlock

• Multiple threads circularly wait for each other

• Example 1
  • Dining philosopher
    this link includes more than what we talked about in class. Only what covered in lectures will be covered in exam)

Example 2:
Thread 1:                  Thread 2:
Lock (A);                  Lock (B);
Lock (B);                  Lock (A);
...                        ...
How to find deadlocks

• If your resource allocation graph (also called resource contention graph or resource dependence graph) has cycle, there is a deadlock
How to avoid deadlocks

• Not using nested locks //using one global lock
• Adding time-out to lock operations
• Following a unified total order for lock acquisition
• …
Distributed systems

Map Reduce

(will not be covered in exam)