

ФОС остаточных знаний по дисциплине «Многопоточное и распределённое программирование»

1. Расположите в порядке увеличения степени абстракции технологии высокопроизводительных вычислений:

- OS Threads, SSE/MMX, C++11 Threads/Java Threads, TBB/Fork-Join framework
- SSE/MMX, OS Threads, TBB/Fork-Join framework, C++11 Threads/Java Threads
- SSE/MMX, OS Threads, C++11 Threads/Java Threads, TBB/Fork-Join framework
- OS Threads, TBB/Fork-Join framework, C++11 Threads/Java Threads, SSE/M

2. Что показывает закон Амдала:

- Прирост производительности в зависимости от доли последовательного кода и числа вычислительных элементов
- Прирост производительности в зависимости от числа вычислительных элементов и используемой технологии распараллеливания
- Число вычислительных элементов, необходимое для достижения заданного уровня роста производительности
- Долю последовательного кода, производительность которого невозможно повысить

3. Как называются функции, внутри которых, в зависимости от используемой технологии, происходит проверка на необходимость завершения работы потока и по результатам проверки возможно выкидывание исключений или завершение потока с применением стека зарегистрированных функций:

- Функции завершения потока
- join-функции
- Cancellation / interruption points
- Signal handlers

4. Чем отличается spin mutex от обычного mutex:

- Остаётся всегда в user space
- Остаётся всегда в kernel space
- Ничем не отличается
- Позволяет читателям производить операции параллельно

5. В каком состоянии находится примитив синхронизации (mutex) в приведённом коде на 1, 3 и 4 строках:

```
0: boost::unique_lock<boost::mutex> lock(mutex);
1:
2: while (messageQueue.empty())
3:     conditionVariable.wait(lock);
```

4:

- Во всех строках захвачен
- Во всех строках освобождён
- **Захвачен в 1 и 4 строках, в 3 на время вызова *wait* освобождается**
- Захвачен в 1 строке, в 3 и 4 освобождён

6. Расположите алгоритмы синхронизации в порядке увеличения потенциальной производительности в жёсткой конкурентной среде:

- **Грубая, тонкая, оптимистичная, неблокирующая**
- Тонкая, грубая, оптимистичная, неблокирующая
- Непрокирующая, оптимистичная, грубая, тонкая
- Грубая, тонкая, неблокирующая, оптимистичная

7. Отметьте вид алгоритмов синхронизации, обычно реализуемый на CAS-операциях и позволяющий добиться максимальной независимости исполнения потоков, гарантируя общий прогресс системы:

- Грубая
- Тонкая
- Оптимистичная
- **Неблокирующая**

8. Выберите корректное определение wait-free класса алгоритмов:

- Любой поток ожидает, пока хотя бы один другой поток не выполнил свою операцию
- Любой поток может выполнить свои функции за конечное число шагов, если ни один другой поток не находится в критической секции
- Система в целом продвигается вперед не зависимо ни от чего
- **Любой поток может выполнить свои функции за конечное число шагов, не зависимо ни от чего**

9. Какая ошибка потенциально может возникнуть в приведённом коде:

```
void f() {
    first_mutex.lock();
    second_mutex.lock();
    ...
}

void g() {
    second_mutex.lock();
    first_mutex.lock();
    ...
}
```

- Гонка данных (Data Race)
- Взаимная блокировка (Deadlock)
- Проблема АВА
- Инверсия приоритетов

10. Почему lock-free алгоритмы легче реализовать на языке со сборщиком мусора, нежели на нативных языках, например C++:

- Невозможна инверсия приоритетов
- Наличие существующих базовых контейнеров в `java.util.concurrent`
- Невозможна проблема АВА
- Наличие CAS-операций

11. Укажите какие виды профилирования желательно применить для комплексного анализа проблем приложения:

- Инструментирующее и сэмплирующее
- Поиск hot spots и анализ издержек на ожидание и синхронизацию (locks and waits)
- Интрузивное и неинтрузивное
- В отладочной сборке и release-сборке

12. Укажите, какой шаблон проектирования позволяет повторно использовать один и тот же вычислительный поток для решения нескольких задач и минимизировать время на создание новых потоков:

- Thread Pool
- Double Check
- Local Serializer
- Pipeline

13. Укажите, какой шаблон проектирования позволяет эффективно организовать потоковую обработку данных:

- Thread Pool
- Double Check
- Recursive Data
- Pipeline

14. В чём заключается одна из основных причин просадки производительности (увеличение времени выполнения операции) при активной конкурентной работе с контейнерами (в предположении, что процессорных ресурсов достаточно):

- Сложность поиска / удаления / вставки элементов
- Переключение контекста процесса
- Потокобезопасность функции new
- Промашки по кэшу процессора

15. Выберите технологию, предоставляющую инструменты работы в терминах задач,

скрывая вычислительные потоки внутри:

- OpenMP
- MPI
- Intel TBB / ForkJoin framework
- Java Threads

16. Выберите технологию, для которой характерна адресация в терминах групп и коммутаторов, с возможностью создания виртуальных топологий:

- OpenMP
- MPI
- Intel TBB / ForkJoin framework
- Java Threads

17. Выберите оптимизацию JIT-компилятора, которая позволяет уменьшить время, затрачиваемое на захват и освобождение примитива синхронизации, не переходя в некоторых случаях в kernel space:

- Объединение захвата примитивов
- Оптимистичный захват
- Адаптивные блокировки
- Замена виртуального вызова

18. Выберите модель памяти, которая гарантирует acquire/release семантику доступа к памяти:

- Sequential Consistency
- Strong-ordered
- Weak-ordered
- Super-weak

19. Выберите тип ввода/вывода, наиболее подходящий для реализации высоко нагруженных серверных частей системы:

- Синхронный блокирующий
- Асинхронный блокирующий
- Синхронный неблокирующий
- Асинхронный неблокирующий

20. Какое консенсусное число CAS-операций:

- 1
- 2
- $2N-2$ (где N-число ячеек памяти, доступных для CAS)
- Бесконечность

21. Как называются функции, внутри которых, в зависимости от используемой технологии, происходит проверка на необходимость завершения работы потока и по результатам проверки возможно выкидывание исключений или завершение потока с применением стека зарегистрированных функций:

- Функции завершения потока
- join-функции
- Cancellation / interruption points
- Signal handlers

22. Чем отличается spin mutex от обычного mutex:

- Остаётся всегда в user space
- Остаётся всегда в kernel space
- Ничем не отличается
- Позволяет читателям производить операции параллельно

23. В каком состоянии находится примитив синхронизации (mutex) в приведённом коде на 1, 3 и 4 строках:

```
0: boost::unique_lock<boost::mutex> lock(mutex);
1:
2: while (messageQueue.empty())
3:     conditionVariable.wait(lock);
4:
```

- Во всех строках захвачен
- Во всех строках освобождён
- Захвачен в 1 и 4 строках, в 3 на время вызова wait освобождается
- Захвачен в 1 строке, в 3 и 4 освобождён

24. Какие различные подходы допускает стандарт POSIX к реализации многопоточности в рамках одного процесса?

- нити в пределах процесса переключаются собственным планировщиком
- переключение между нитями осуществляется ядром системы
- процессу выделяют некоторое количество системных нитей, но он имеет собственный планировщик
- переключение между нитями осуществляется внешними сигналами

25. При гибридной реализации многопоточности, количество пользовательских нитей в процессе:

- не может превосходить количество системных нитей
- может превосходить количество системных нитей
- всегда равно количеству системных нитей

26. Если какая-то из пользовательских нитей процесса исполняет блокирующий системный вызов, то:

- **блокируется весь процесс**
- блокируется эта нить
- блокировки нити и процесса не происходит

27. Процесс взаимодействует с ядром операционной системы при помощи:

- **СИСТЕМНЫХ ВЫЗОВОВ**
- потоков
- нитей

28. При исполнении системного вызова, процесс исполняет специальную команду, которая:

29.

- переключает адресное пространство, но не передает управление ядру
- **переключает адресное пространство и передает управление ядру**
- передает управление ядру, но не переключает адресное пространство

30. Нарушения целостности данных одного процесса приводят:

- **к аварийному завершению этого процесса, но не затрагивают другие процессы**
- к аварийному завершению всех процессов
- к аварийному завершению других процессов, но не затрагивают этот процесс

31. Определите класс по умолчанию для переменной numt OpenMP:

```
int i=0;int numt = omp_get_max_threads();#pragma omp parallel forfor(i=0; i< numt; i++)
Work(i);
```

- threadprivate
- private
- **shared**

32. Найдите ошибку в следующем фрагменте программы OpenMP:

```
#define N 10int A[N],B[N];#pragma omp parallel default(shared) num_threads(10){ int
iam=omp_get_thread_num(); int tmp; tmp=A[iam]; B[iam]=tmp;}
```

- чтение/изменение переменной tmp выполняется без какой-либо синхронизации
- изменение общей переменной B[iam] выполняется без какой-либо синхронизации
- **в данном фрагменте программы ошибки нет**

33. Способ распределения витков цикла между нитями группы задается при помощи клаузы schedule(<алгоритм планирования>[,<число итераций>]).
Найдите ошибку в следующем фрагменте программы OpenMP:

```
#pragma omp parallel default(shared){ int i; #pragma omp for schedule(dynamic,
omp_get_thread_num()) for (i=0; i<n; i++) { work(i); }}
```

- значение параметра <число итераций> клаузы schedule отличается для каждой нити группы

- в данном фрагменте программы ошибки нет
- при динамическом планировании, задаваемом клаузой schedule(dynamic), параметр <число итераций> задать нельзя

34. Выберите наилучшую стратегию распределения витков цикла между нитями, которая для следующего фрагмента программы OpenMP даст минимальное время выполнения:

```
#include <omp.h>#include <unistd.h>#define msec 1000int main (void){
omp_set_num_threads (8); #pragma omp parallel { #pragma omp for schedule (runtime)
for(int i=0; i<80;i++) { sleep (msec); } }}
```

- export OMP_SCHEDULE="static,15"
- export OMP_SCHEDULE="static,20"
- export OMP_SCHEDULE="static,10"

35. Клауза copyin OpenMP:

- может быть использована только для переменных, указанных в клаузе private
- может быть использована только для переменных, указанных в директиве threadprivate
- может быть использована как для переменных указанных в директиве threadprivate, так и для переменных, указанных в клаузе private

36. Определите значение переменной team_size по завершении выполнения следующей программы OpenMP:

```
#include <stdio.h>#include «»omp.h»»int main(){ int team_size; team_size=0; #pragma
omp parallel num_threads(2) { if (omp_get_thread_num() == 0)
{team_size=omp_get_team_size(omp_get_level ()); } } printf(«»Team
Size=%d\n»»,team_size);}
```

- 0
- 2
- 1

37. Директива master OpenMP

- определяет блок операторов в программе, который будет выполнен одной нитью группы. Остальные нити группы дожидаются завершения выполнения этого блока
- определяет блок операторов в программе, который будет выполнен master-нитью. Остальные нити группы не дожидаются завершения выполнения этого блока
- определяет блок операторов в программе, который будет выполнен master-нитью. Остальные нити группы дожидаются завершения выполнения этого блока

38. Выберите наилучшую стратегию распределения витков цикла между нитями, которая для следующего фрагмента программы OpenMP даст минимальное время выполнения:

```
#include <omp.h>#include <unistd.h>#define msec 1000int main (void){
omp_set_num_threads (8); #pragma omp parallel { #pragma omp for schedule (runtime)
for(int i=0; i<100;i++) { sleep ((100-i)*msec); } }}
```

- export OMP_SCHEDULE="static"
- export OMP_SCHEDULE="dynamic"
- export OMP_SCHEDULE="static,10"

39. Пусть следующая программа скомпилирована компилятором, поддерживающим вложенный параллелизм.

```
#include <stdio.h>#include <omp.h>int counter;int main(){ counter=0;
omp_set_nested(1); #pragma omp parallel num_threads(2) { if (omp_get_thread_num() ==
0) { #pragma omp parallel num_threads(2) { #pragma omp atomic counter++; } } }
printf("Counter=%d\n",counter);}
```

Определите значение переменной counter по завершении выполнения этой программы:

- 4
- 2
- 1

40. При реализации компилятором редукционного оператора, описанного при помощи клаузы reduction (+: sum), где переменная sum имеет тип integer, для каждой нити создается локальная копия переменной sum, начальное значение которой будет инициализировано:

- MAXINT (максимально возможное целое число)
- -MAXINT (минимально возможное целое число)
- 0

41. Найдите ошибку в следующем фрагменте программы OpenMP:

```
#define N 10int A[N],B[N],tmp;#pragma omp parallel default(shared) num_threads(10){ int
iam=omp_get_thread_num(); tmp=A[iam]; B[iam]=tmp;}
```

- чтение/изменение общей переменной tmp выполняется без какой-либо синхронизации
- в данном фрагменте программы ошибки нет
- изменение общей переменной B[iam] выполняется без какой-либо синхронизации

42. Процессорная модель консистентности памяти определяется следующим условием:

- все процессы наблюдают операции записи любого процесса в порядке их выполнения. Операции записи различных процессов могут наблюдаться разными процессами в разном порядке

- все процессы наблюдают операции записи любого процесса в порядке их выполнения. Для каждой переменной существует общее согласие относительно порядка, в котором процессы модифицируют эту переменную. Операции записи различных процессов могут наблюдаться разными процессами в разном порядке
- все процессы наблюдают все обращения к ячейкам памяти в одном и том же порядке. Обращения не упорядочены по времени

43. Найдите ошибку в следующем фрагменте программы OpenMP:

```
#include <omp.h>int main (void){ #pragma omp parallel { int numt; #pragma omp single
numt=omp_get_num_threads(); if (numt < 4) do_small_work(); else do_big_work (); }}
```

- после конструкции single отсутствует директива barrier
- в данном фрагменте программы ошибки нет
- в директиве single отсутствует клауза copyprivate(numt)

44. Пусть следующая программа скомпилирована компилятором, поддерживающим вложенный параллелизм.

```
#include <stdio.h>#include <omp.h>int counter;int main(){ counter=0;
omp_set_nested(0); #pragma omp parallel num_threads(2) { #pragma omp parallel
num_threads(2) { #pragma omp atomic counter++; } } printf(«Counter=%d\n»,counter);}
```

Определите значение переменной counter по завершении выполнения этой программы:

- 1
- 4
- 2

45. Найдите ошибку в следующем фрагменте программы OpenMP:

```
#define N 10int A[N], sum;#pragma omp parallel default(shared) num_threads(10){ int
iam=omp_get_thread_num(); #pragma omp critical (update_a) #pragma omp critical
(update_a) sum +=A[iam];}
```

- дедлок — взаимная блокировка нитей, возникающая в результате того что одноименные критические секции вложены друг в друга
- в данном фрагменте программы ошибки нет
- критические секции не могут быть вложены друг в друга

46. Найдите ошибку в следующем фрагменте программы OpenMP:

```
int main (void){ int a, i; #pragma omp parallel shared(a) private(i) { #pragma omp master a
= 0; #pragma omp for reduction(+:a) for (i = 0; i < 10; i++) { a += i; } }}
```

- в данном фрагменте программы ошибки нет
- в директиве parallel клауза shared(a) должна быть заменена на private(a)
- перед директивой for отсутствует директива barrier

47. Функция omp_get_num_threads возвращает:

- максимально возможное количество нитей, которые могут быть использованы при выполнении всей программы
- номер нити в группе
- количество нитей в текущей параллельной области

48. Параллельная область в OpenMP создается при помощи:

- директивы `parallel`
- вызова функции `omp_set_max_active_levels`
- вызова функции `omp_set_num_threads`

49. Выберите наиболее подходящую оптимизацию, которая позволит сократить время выполнения следующего фрагмента программы:

```
#include <omp.h>#include <unistd.h>#define msec 1000int main (void){ int i;
omp_set_num_threads (8); #pragma omp parallel for for (i=0; i<80; i++) sleep (msec);
#pragma omp parallel for for (i=0; i<80; i++) sleep (msec);}
```

- для первого цикла, выполнение витков которого распределяется между нитями при помощи директивы `for`, добавить клаузу `schedule(dynamic)`
- для второго цикла, выполнение витков которого распределяется между нитями при помощи директивы `for`, добавить клаузу `schedule(dynamic)`
- объединить две подряд стоящие параллельные области в одну

50. Найдите ошибку в следующем фрагменте программы:

```
#define N 10int A[N],B[N];#pragma omp parallel default(shared){ int i;....#pragma omp
master for (i=0; i<N; i++) { A[i]=0; } #pragma omp for for (i=0; i<N; i++) B[i]=A[i];}
```

- оператор `for` не может быть использован внутри конструкции `master`
- по завершении конструкции `master` отсутствует директива `barrier`
- в данном фрагменте программы ошибки нет

51. При реализации компилятором редукционного оператора, описанного при помощи клаузы `reduction (*: prod)`, где переменная `prod` имеет тип `integer`, для каждой нити создается локальная копия переменной `prod`, начальное значение которой будет инициализировано:

- 1
- MAXINT (максимально возможное целое число)
- 0