АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

РАЗДЕЛ 3.1

СИНХРОНИЗАЦИЯ ПРОЦЕССОВ

ОСНОВНЫЕ ПОНЯТИЯ

Активность – совокупность действий (операций), направленная на достижение некоторой цели.

P: a,b,c , Q: d,e,f – активности из атомарных операций

PQ: a,b,c,d,e,f – последовательное выполнение активностей **P,Q**

Псевдопараллельное исполнение активностей **(interliving)** дает следующие варианты совместного выполнения активностей **P,Q**

a,b,c,d,e,f

a,b,d,c,e,f это всевозможные варианты чередования атомарных операций

a,b,d,e,c,f при условии сохранения порядка выполнения внутри **P** и **Q**

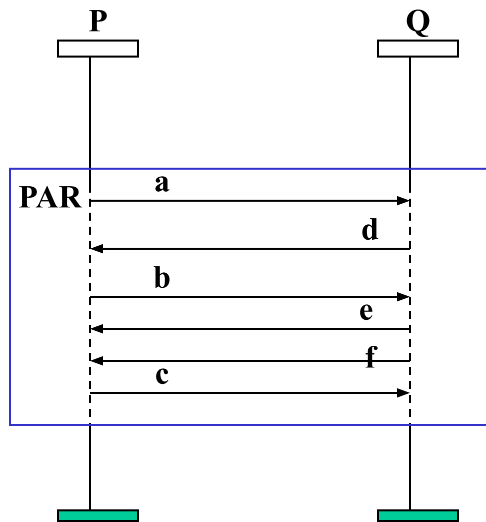
….

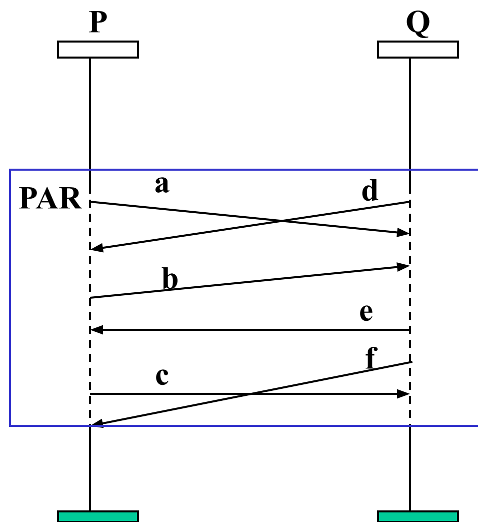
d,e,f, a,b,c

Детерминированный набор активностей – дает при псевдопараллельном исполнении на одном и том же наборе входных данных дает одинаковый результат, в противном случае он **Недетерминирован** и может давать разные результаты

Например, P: x=2, y=x-1 Q: x=3, y=x+1 при псевдопараллельном исполнении могут дать результаты (x,y): (3,4), (2,1), (2,3), (3,2)

Хотим до получения результата узнать является ли набор активностей недетерминированным


P
Q
PAR
a
d
b
e
f
c


P
Q
PAR
a
d
b
e
f
c

УСЛОВИЯ БЕРНСТАЙНА

Достаточные условия Бернстайна

Пусть R(P) – набор входных переменных активности - объединение наборов входных переменных для всех ее неделимых (атомарных) действий,
W(P) – набор выходных переменных активности как объединение наборов выходных переменных для всех неделимых действий.

Для P: x=a+b, Q: y=x*d   R(P)=(a,b,x,d) и   W(P)=(x,y),   а R(P)&W(P)≠0 - непусто

Бернстайн: Выполнение P и Q детерминировано, если
- пересечение **W(P)&W(Q)=0** (пусто)| P: x=a+b, R(P)={a,b}, W(P)={x}
- пересечение **W(P)&R(Q)=0** (пусто)| Q: y=x*d, R(Q)={x,d},W(Q)={y}
- пересечение **R(P)&W(Q)=0** (пусто) | W(P)& W(Q)=0, W(P)&R(Q)≠0, R(P)&W(Q)=0
Если эти условия не соблюдены, то при параллельном выполнении м.б недетерминизм

Условия распространяются на **N** процессов

Условия Бернстайна слишком жестки, они гарантируют детерминизм практически невзаимодействующим процессам

Чтобы обеспечить параллельное выполнение надо ограничить возможности **interliving** (чередования**),** что и достигается **синхронизацией**

ТРЕБОВАНИЯ К АЛГОРИТМАМ ОРГАНИЗАЦИИ ПРОЦЕССОВ

Пять условий хорошего алгоритма организации взаимодействия процессов:

1. Задача решается на компьютере, не имеющим специальных команд взаимоисключения, но основные операции которого - load, store, test – атомарны
2. Не должно быть ограничений на относительные скорости выполнения процесса или на число процессоров, на которых выполняются процесс
3. Если процесс Pi исполняется в своем критическом участке, то не д.б. никаких других процессов , которые выполняются в соответствующих критических секциях (условие взаимоисключения - **mutual execution**)
4. Процессы, не находящиеся в своих критических участках и не претендующие на доступ к ним не имеют права препятствовать другим процессам входить в собственные критические участки.

Принятие решения о порядке входа в критическую секцию осуществляется процессами, которые находятся перед критическим участком (а не в секции эпилога, где довыполняются операции после выхода из критической секции) и такое решение не должно вырабатываться слишком долго (условие прогресса - **progress**)

5. Ожидание процесса от момента запроса доступа до входа в свою критическую секцию должно быть ограничено – другие процессы могут пройти за это время свои критические участки лишь конечное число раз (условие ограниченного ожидания – **bound waiting**).

АЛГОРИТМ – ЗАПРЕТ ПРЕРЫВАНИЙ

Наиболее простой подход

```
while (some condition) {
    запретить-все-прерывания
        critical section
    разрешить-все-прерывания
        remainder section
}
```

Выход процесса из состояния исполнения без завершения осуществляется по прерыванию, то внутри критической секции никто не может вмешаться в работу процесса.

Однако, такое **решение чревато последствиями, если процесс в критическом участке зациклится или умрет**.

АЛГОРИТМ – ПЕРЕМЕННАЯ-ЗАМОК

Используем переменную-замок с начальным значением **true**, доступную всем процессам. Процесс может войти в критическую секцию, если замок открыт (равен **true**) и одновременно закрыть замок присвоить переменной **false**. При выходе замок открываем, присваивая ему true.

```
shared bool lock=true /*shared – описатель разделяемой переменной*/
while (some condition) {
    while(lock){lock=false;
        critical section
    lock=true;
       remainder section}
}
```

Такое **решение не обеспечивает условия взаимоисключения**, поскольку **действие while(lock); lock=false; не атомарно**, а значит после процесса P1, получившего доступ по lock=true, проверить while до присвоения lock=false сможет другой процесс P2 и тоже войти в критическую секцию

АЛГОРИТМ – СТРОГОЕ ЧЕРЕДОВАНИЕ

Используем общую переменную с начальным значением 0, доступную двум процессам. Переменная будет явно указывать на процесс, которому разрешен вход в критический участок. Для процесса **Pi** получим :

```
shared int turn=0;
while (some condition) {
    while (turn = i) {
        critical section
    turn=1-i;
       remainder section}
}
```

Решение обеспечивает условия взаимоисключения, процессы входят в критическую секцию строго по очереди: **P0, P1, P0, P1, P0,** …

Но **алгоритм не удовлетворяет условию прогресса**. Например, если значение **turn=1** и процесс **P0** готов войти в критический участок, то он не сможет это сделать пока процесс **P1** не изменит значение **turn** даже, если это произойдет в **remainder section**

АЛГОРИТМ – ФЛАГИ ГОТОВНОСТИ

Недостаток предыдущего алгоритма в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Исправим эту ситуацию, используя разделяемый массив флагов готовности входа процессов в критический участок:

shared int ready[2]=(0,0);

Процесс **P**i когда входит в критическую секцию, то присваивает флагу ready[i] значение 1, а после выхода из критической секции значение 0. Процесс не входит в критическую секцию, если другой процесс уже готов к входу или находится в ней.

```
while (some condition) {
     ready[i]=1;
     while(ready[1-i]) {
          critical section
    ready[i]=0;
        remainder section }}
```

Решение обеспечивает условия взаимоисключения, процессы входят в критическую секцию после завершения эпилога в ранее обслуживаемом. Но **алгоритм не удовлетворяет условию прогресса**. Если после выполнения ready[0]=1 для процесса P0 и монитор ОС передал управление P1, а он выполнил ready[1]=1, то два процесса оказываются в deadlock на входе в критическую секцию.

АЛГОРИТМ ПЕТЕРСОНА

Пусть оба процесса имеют доступ к разделяемым массиву флагов готовности и переменной очередности:

```
shared int ready[2]=(0,0);
shared int turn;
while (some condition) {
     ready[i]=1;
     turn=1-i;
     while(ready[1-i] && turn == 1-i) {
          critical section
    ready[1-i]=0;
        remainder section }}
```

При исполнении пролога критической секции процесс **Pi** заявляет о своей готовности выполнить критический участок и одновременно предлагает другому процессу приступить к его выполнению.

Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда следует после другого. Тем самым работу в критическом участке выполнит процесс, принявший последнее предложение. **Алгоритм дает корректное решение для 2 процессов.**

АЛГОРИТМ БУЛОЧНОЙ (Bakery algorithm)

Алгоритм для **n** взаимодействующих процессов. Идея алгоритма: Каждый вновь прибывающий процесс (клиент) получает талончик на обслуживание с номером. Процесс с наименьшем номером на талончике обслуживается следующим. Алгоритм из-за неатомарности операции вычисления следующего номера не гарантирует, что у всех процессов будут талончики с разными номерами. Разделяемые структуры данных – два массива:

shared choosing[n]={false,…false}, shared int number[n]={0,…0};

Структура алгоритма для процесса Pi:

```
while (some condition) {
    choosing[i]=true;
    number[i]=max(number[0], … number[n-1])+1; //получение талончика
     …
    for (j=0; j<n; j++) {
        while(choosing[j]);
        while(number[j] !=  0 && (number[j], j) < (number[i], i)) } //пропуск мл.ном
         critical section
   number[j]=0; choosing[j]=false;
       remainder section
}
```

АЛГОРИТМ БУЛОЧНОЙ (продолжение)

,где

(a,b) < (c,d), если a < c или если a == c и b < d

max(a0, a1, … an) – это число k такое, что k >= ai для всех i = 0, …, n

Аппаратная поддержка взаимоисключений

Test-and-Set – неделимая команда проверить и присвоить

Ее неделимый алгоритм:

```
int Test-and-Set (int *target) {
     int tmp=*target
    *target=1;
    return tmp; }
```

Swap – обменять значения

```
void Swap (int *a, int *b) {
int tmp=*a;
*a=*b
*b=tmp }
```

Мьютекс

это простейший двоичный семафор, который может находиться в одном из двух состояний — отмеченном или неотмеченном. Когда какой-либо поток, принадлежащий любому процессу, становится владельцем объекта mutex, последний переводится в неотмеченное состояние. Если задача освобождает мьютекс, его состояние становится отмеченным.

Реализация критической секции через мьютекс

```
#include <pthread.h>
typedef pthread_mutex_t CRIT_SECTION;
```

Создадим прототипы функций для работы с критической секцией

```
void WTF_InitCritSect(CRIT_SECTION *);
void WTF_EnterCritSect(CRIT_SECTION *);
void WTF_LeaveCritSect(CRIT_SECTION *);
void WTF_DestroyCritSect(CRIT_SECTION *);
```

Реализация функциональности прототипов

```
void WTF_InitCritSect(CRIT_SECTION * cs) {
	pthread_mutex_init(cs, NULL);
}

void WTF_EnterCritSect(CRIT_SECTION * cs) {
	pthread_mutex_lock(cs);
}

void WTF_LeaveCritSect(CRIT_SECTION * cs) {
	pthread_mutex_unlock(cs);
}

void WTF_DestroyCritSect(CRIT_SECTION * cs) {
	pthread_mutex_destroy(cs);
}
```

Реализация синхронизации в Java:

```
static class WTF_Lock extends Object {}
static public WTF_Lock lockObject = new WTF_Lock();
…

// Thread 1:
synchronized (lockObject) {…}
…

// Thread 2:
synchronized (lockObject) {…}
…
```

Пример для C++(1):

```
// Библиотека потоков POSIX
#include <pthread.h>
typedef pthread_mutex_t CRIT_SECTION;
....
// Функции для работы со стандартными типами
void WTF_InitCritSect(CRIT_SECTION * cs)
    {pthread_mutex_init(cs, NULL); }
void WTF_EnterCritSect(CRIT_SECTION * cs)
    {pthread_mutex_lock(cs); }
void WTF_LeaveCritSect(CRIT_SECTION * cs)
    {pthread_mutex_unlock(cs); }
void WTF_DestroyCritSect(CRIT_SECTION * cs)
    {pthread_mutex_destroy(cs); }
```

Пример для С++(2):

```
void main (void) {
    pthread_t task1, task2, task3;
    ...
    //創здпние потоков, которые будут бороться за ресурс
    pthread_create (&task1, NULL, TaskThread, (void*) &thread1Info);
    pthread_create (&task2, NULL, TaskThread, (void*) &thread2Info);
    pthread_create (&task3, NULL, TaskThread, (void*) &thread3Info);
    pthread_detach (task1);  //независимое исполн потоков
    pthread_detach (task2);
    pthread_detach (task3);
    … }
```

Пример для С++(3):

```
void * TaskThread (void* threadInfo) {
  // Код, конфигурирующий работу потока согласно
  // параметрам, переданным через threadInfo.
  // Например, N – объём ресурса, необходимый потоку
  ...
  // Основной цикл работы, в котором
  // происходит обращение к ресурсу
  while (1) {   ...
    PrinterMonitor::instance().acquirePages (N); // Блокирующий запрос ресурса
    // Процесс что-то печатает на страницах
    PrinterMonitor::instance().addPaperToTray (N); // Процесс освобождения
                                                    ресурса
   ...   }
  ...}
```

// Паттерн Singleton **Пример для C++(4):**

```
const int gPagesTotal = 10; // Общий объём ресурса
class PrinterMonitor {// Монитор ресурса
    int pagesLeft;
      CRIT_SECTION lock; // объект для синхронизации
    …
    public:
    static PrinterMonitor instance () {…} // Получение объекта монитора
    void addPaperToTray (int N) {    // Освобождение ресурса
     pagesLeft += N;          // Освободить N страниц
    }
```

Пример для С++(5):

```
// Блокирующий захват ресурсов
void acquirePages (int N) {
        WTF_EnterCritSect (& lock);
      // Ждать, пока освободится N страниц
      while (pagesLeft < N) {}
    // Захватить нужный объём ресурса
      pagesLeft -= N;
        WTF_LeaveCritSect (& lock);
 }
}
```

static class WTF_Lock extends Object {} **Пример для Java(1):**

```
static public WTF_Lock lockObject = new WTF_Lock();
…
package chapt14;
// Класс монитора ресурсов, также Singleton
public class PrinterMonitor {
    int pagesLeft = 10;
    WTF_Lock lock = new WTF_Lock();
    ...
    public static PrinterMonitor instance () {…}
    public void addPaperToTray (int N) {
        // Освободить ресурс – аналогично добавлению
        //N страниц в лоток принтера
        pagesLeft += N;        }
```

Пример для Java(2):

```
// Блокирующий захват ресурса
public void acquirePages (int N) {
  synchronized (WTF_Lock) {
  // Ждать, пока освободится N страниц
  while (pagesLeft < N) {}
  // Захватить нужный объём ресурса
  pagesLeft -= N;
    }
  }
}
```

Пример для Java(3):

```
public class SomeThreads {
 public static void main(String args[]) {
    final StringBuffer s = new StringBuffer();
    // Запуск потоков, использующих ресурс
    new Thread() {
       public void run() {
           // Блокирующий запрос ресурса
           PrinterMonitor::instance().acquirePages (N);
           // Использование ресурса
           // Процесс освобождения ресурса
           PrinterMonitor::instance().addPaperToTray (N);
       }
    }
```

Пример для Java(4):

```
.start();
// Ещё один аналогичный поток
new Thread() {
 public void run() {
        ...
        PrinterMonitor.instance().acquirePages (N);
          // Process doing some printing
        PrinterMonitor.instance().addPaperToTray (N);
         ...
        }
    }
.start();
...
}
```

Мьютекс (Mutex) для Java

Это простейший двоичный семафор, который может находиться в одном из двух состояний – отмеченном или неотмеченном.
Когда какой-либо поток, принадлежащий любому процессу, становиться владельцем объекта «мьютекс», последний переводится в неотмеченное состояние (ресурс захвачен). Если задача освобождает мьютекс, его состояние становится отмеченным (ресурс освобожден).

Реализация синхронизации в Java

```
static Semaphore mutex = new Semaphore(1);
…
//consumer thread
void run() {
    …
    //critical section
    mutex.acquire();
    …
    mutex.release();
    …
}
```

static Semaphore mutex = new Semaphore(1)
Мьютекс реализован при помощи семафора (с java 1.5).. Он создан за счет того, что в конструкторе передано одно разрешение (permit), семафор может захватить только один поток

mutex.acquire(); - захват семафора потоком
mutex.release(); - освобождение семафора потоком

Паттерн Singleton

Это паттерн проектирования, который гарантирует, что в приложении будет единственный экземпляр класса с глобальной точкой доступа.

Данный паттерн применяется, когда:

1. В системе должно существовать не более одного экземпляра заданного класса.
2. Экземпляр должен быть легко доступен для всех клиентов данного класса.

Существует множество подходов к реализации паттерна по способу инициализации:

- “Не ленивая инициализация” – единственный экземпляр класса создается в тот же момент, как создается объект класса

- “Ленивая инициализация “– единственный объект создается только при первом его запросе

Не ленивая инициализация упрощает создание объекта в многопоточных системах (так как создает его до первого вызова), а **ленива**я инициализация позволяет создать объект только когда он потребуется впервые в системе, что уменьшает нагрузку на память

Singleton. Enum

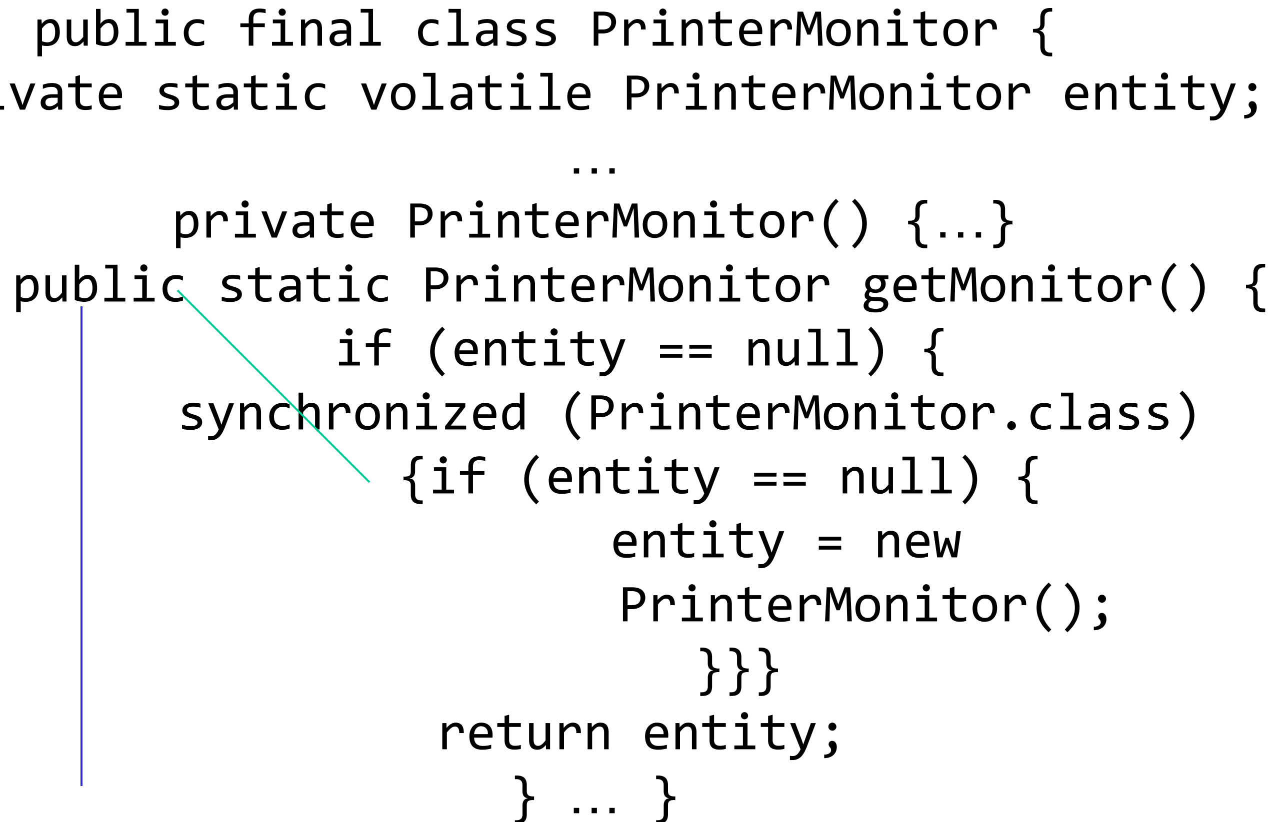
```
public enum SingletonEnum {
        //Singleton object
            INSTANCE;
                ..
              }

        //get singleton
   SingletonEnum.INSTANCE;
```

Enum - Перечисляемый тип, чьё множество значений представляет собой ограниченный список идентификаторов. Поддерживает многопоточность, так как экземпляр **enum** является **public final static volatile** переменной.

Singleton. Double Checked Locking & volatile

Пример реализации синглтона, с использованием ленивой инициализации. По этой причине требуется конструктор с критической секции, при работе в многопоточном приложение.



```
public final class PrinterMonitor {
private static volatile PrinterMonitor entity;
...
private PrinterMonitor() {...}
public static PrinterMonitor getMonitor() {
    if (entity == null) {
    synchronized (PrinterMonitor.class)
        {if (entity == null) {
            entity = new
            PrinterMonitor();
        }}}
    return entity;
} ... }
```

Singleton. Double Checked Locking & volatile

Реализация называется **Double Checked Locking** по причине двойной проверка при создании объекта.

Проверка в блоке synchronized повторяется по причине того, что перед данной критической секцией могут остановиться 2 и более потока исполнения.
Но объект синглтона должен создать только первый зашедший в эту секцию поток. Остальные потоки должны получить указатель на уже существующий объект.
private static volatile PrinterMonitor entity;
Модификатор volatile используется для того, чтобы в случае многоядерной системы данная переменная создавалась и хранилась в общей памяти, а не в личной памяти одного из ядер (так как такая ситуация может привести к тому, что будет создано несколько экземпляров синглтона, что не соответствует ожидаемому поведению)

Singleton. Use example (1)

Полный код с создание класса синглтона (**PrinterMonitor**), пользующегося разделяемым ресуксом (**Worker**).

```
final class PrinterMonitor {
vate static volatile PrinterMonitor entity;
private int pagesLeft;
private Lock lockObtain, lockRelease;
//private constructor
private PrinterMonitor() {
pagesLeft = 10;
lockObtain = new ReentrantLock(true);
lockRelease = new ReentrantLock(true);
}
ublic static PrinterMonitor getMonitor()
{ … }
```


Закрытый конструктор синглтон. Внутри него создаются локи, которые будут управлять ресурсом один на выделение ресурса, другой на получение разделяемого ресурса - для поддержания атомарности операции получения.

Reentrant lock – Лок на вхождение. Только один поток может зайти в защищенный блок. Класс поддерживает «честную» (fair) и «нечестную» (non-fair) разблокировку потоков. При «честной» разблокировке соблюдается порядок освобождения потоков, вызывающих lock(). При «нечестной» разблокировке порядок освобождения потоков такая разблокировок не гарантируется, но работает быстрее. По умолчанию, используется «нечестная» .

Лок лучше блока synchronized:

- Взятие и освобождение лока может происходить в разных методах
- При запросе получения лока можно задавать условия ожидания чтобы не было мертвой блокировки потока (условия – время, прерывания или запроса на получение с возвращение результата в виде bool переменной)

Singleton. Use example (2)

```
//resource release
public void addPaperToTray(int value) {
    try {
        lockRelease.lockInterruptibly();
        pagesLeft += value;
    } catch (InterruptedException e) { … }
    finally { lockRelease.unlock(); }
}
//resource capture
public void acquiredPages(int value) {
    try {
        lockObtain.lockInterruptibly();
        while (pagesLeft < value) {}
        pagesLeft -= value;
    } catch (InterruptedException e) { … }
    finally { lockObtain.unlock(); }
}
```

Методы захвата (**resource capture**) и освобождения ресурса (**resource release**) с использование локов, для выделения критических секций.

Singleton. Use example (3)

Код класса, который использует разделяемый ресурс. Наследуется от интерфейса **Runnable,** чтобы запускаться в отдельном потоке.

```
//processing thread
public class Worker implements Runnable {
…
@Override
public void run() {
…
//get resourcers
PrinterMonitor.getMonitor().acquiredPages(value);
…
//release resources
PrinterMonitor.getMonitor().addPaperToTray(value);
} … }
```

Singleton. Use example (4)

```
public class TestMonitor {
public static void main(String[] args) {
        int numberOfThreads = 5;
        Thread[] workers = new
    Thread[numberOfThreads];
   for (int i = 0; i < numberOfThreads;
              i++) {
               //create thread
        workers[i] = new Thread(new
           Worker());
          //start thread (start run method)
           workers[i].start();
                 }
               }
             }
```

Пример кода запуска потоков.