Prof. Dr. Th. Letschert
FB MNI

9. Juni 2002

Übung 4 - Betriebssysteme I

Aufgabe 1

1. Erläutern Sie die Begriffe der “transparent” und der “virtuell” mit ihrer in der Informatik üblichen Bedeutung.
2. Wie werden in der Informatik die Größen 2^{10}, 2^{20} und 2^{30} üblicherweise bezeichnet?
3. Welche Speicherstellen können von der CPU direkt über den Adressbus angesprochen werden? Erläutern Sie kurz aus welchen Komponenten der reale Speicher besteht.
4. Erläutern Sie kurz folgende Begriffe:
   - physische Adresse
   - logische Adresse
   - physischer Adressraum
   - logischer Adressraum
   - virtuelle Adresse
   - virtueller Speicher
5. Von wem werden wann absolute, relozierbare und virtuelle Adressen erzeugt und von wem werden diese dann wann (falls notwendig) in reale Speicheradressen umgesetzt?
6. Ist ein System mit virtuellen Adressen automatisch auch eins mit virtuellem Speicher?
7. Wie ist es umgekehrt: Ist ein System mit virtuellem Speicher auch automatisch eins mit virtuellen Adressen?
8. In welcher Beziehung stehen statische/dynamische Bindung und virtuelle Adressen:
   - Kann/muss ein statisch gebundenes Programm virtuelle Adressen enthalten?
   - Kann/muss ein dynamisch gebundenes Programm virtuelle Adressen enthalten?
9. Was bestimmt die Größe des logischen, physischen und virtuellen Adressraums. Erläutern Sie am Bespiel eines PCs mit Pentium–Prozessor und 64 MB RAM.
10. Was ist ein virtueller Adressraum und was ist ein physischer Adressraum? Welche der folgenden Einheiten hat – falls überhaupt – welche Art von Adressraum:
    - ein Thread,
    - ein Prozess,
    - eine Anwendung,
    - eine Rechnerarchitektur,
    - ein bestimmter Rechner,
    - ...

11. Was ist ein virtueller Speicher und was ist ein physischer Speicher? Welche der folgenden Einheiten hat – falls überhaupt – welche Art von Speicher:

    - ein Thread,
    - ein Prozess,
    - eine Anwendung,
    - eine Rechnerarchitektur,
    - ein bestimmter Rechner,
    - ...

12. Was passiert, wenn der logische Adressraum eines Prozesses voll ist? Kann es dazu überhaupt kommen?

13. Was passiert, wenn der virtuelle Speicher voll ist? Ist das möglich? Was bedeutet überhaupt die Aussage "der virtuelle Speicher ist voll", was ist der "virtuelle Speicher"?

14. Machen virtuelle Adressen/virtueller Speicher die Implementierung von gemeinsamen Bibliotheken eher schwieriger oder eher leichter?

Aufgabe 2

1. In welchem Adressraum agieren die C++–Operatoren `new` und `delete`?

2. Müssen `new` und `delete` *Thread–save* sein, d.h. müssen sie mit Synchronisationsmitteln auf "gleichzeitigen" Aufruf durch Threads vorbereitet sein? Erläutern Sie die möglichen (Synchronisations–) Probleme und entweder deren Lösung oder den Grund warum sie nicht auftreten können. Wo werden die Synchronisationsprobleme – falls notwendig – gelöst: in der C++–Bibliothek, an anderer Stelle?

3. Müssen `new` und `delete` *Process–save* sein, d.h. müssen sie mit Synchronisationsmitteln auf "gleichzeitigen" Aufruf durch Prozesse vorbereitet sein?

4. Die C++–Operatoren `new` und `delete` sind freie Funktionen, die sich an eine Heap–Verwaltung richten. Konzeptionell kann man sie also etwa wie folgt darstellen:

   ```
   void * operator new ()           { heap.allocate(); }
   void   operator delete (void *p) { heap.terminate(p); }
   ```

   Wobei `heap` eine Instanz einer Klasse `Heap` ist, die den Heapspeicher und seine Verwaltung umfasst. Wieviele Instanzen von `Heap` gbt es: Eine pro Thread, pro Prozess, pro System, ... ?

5. Was passiert bei einem `new` und einem `delete` bei moderen System mit virtuellem Speicher und Seitenverfahren. Erläutern Sie kurz die durch diese Aufrufe ausgelösten Aktionen.

Aufgabe 3

1. Betrachten Sie folgendes Programm:

```
#include <iostream>

typedef unsigned int uint;

int main () {
  char *a = new char;
  cout << " a = " << hex << uint(a) << endl;
  char * d = new char[0x1000];
  cout << " d = " << hex << uint(d) << "..." << uint(d)+0x1000 <<endl;
  int i = 0;
  while ( true ) {
    *a = 0;
    cout << dec << i << ". belegt Adresse " << hex << uint(a) <<endl;
    ++a;
    ++i;
  }
}
```

2. Welche Art von Adressen gibt das Programm aus: logische, virtuelle, physische, reale, abstrakte, ...?

3. Bei welchem Wert von `a` wird das Programm abstürzen: Nach der ersten Erhöhung, bei der Zuweisung nach der ersten Erhöhung, wenn `a` den Wert von `d` erreicht hat, wenn `a` den Wert `d + 0x1000` erreicht hat, wenn der physische Speicher voll ist, wenn der virtuelle Speicher voll ist, wenn der Adressraum des Prozesses voll ist, ... ?

4. Schreiben Sie ein kleines Testprogramm um festzustellen in welche Richtung Stack und Heap wachsen.

5. Unter “Der virtuelle Speicher ist voll!” kann man verstehen, dass weitere Speicheranforderungen der Prozesse nicht mehr befriedigt werden können, weil der Swap–Bereich voll ist. Ist dies ein so seltenes Ereignis, dass das Betriebssystem in dem Fall guten Gewissens abstürzen kann?

6. Folgendes Programm versucht den Speicher zu füllen. Welchen Speicher? Was passiert wenn er voll ist? Kommt es dazu?

```
#include <iostream>

typedef unsigned long uint;

struct A {
  char a[0x1000]; // (Einheiten von 4 kB)
};

void fStack (uint &i) {
  A a;
  cout << i << ". im Stack an Pos: " << &a << endl;
  fStack( ++i );
}

int main () {
  uint stackFrames = 0;
  fStack(stackFrames);
}
```

Experimentieren Sie!

Aufgabe 4

1. In welchem Maß führen Segmentierung und Paging zu interner und/oder externer Fragmentierung? Was ist das überhaupt?
2. Was ist eine “Seite” und was ein “Seitenrahmen”?
3. Was ist eine Seitentabelle, welche Informationen enthält sie, wer benutzt sie wozu, wann wird sie verändert bzw. gewechselt?
4. Was ist eine Rahmentabelle, welche Informationen enthält sie, wer benutzt sie wozu, wann wird sie verändert bzw. gewechselt?
5. Was ist eine Seitendeskriptor, welche Informationen enthält er, wer benutzt ihn wozu, wann wird er verändert bzw. gewechselt?
6. Was ist ein Seitenfehler (*page fault*)? Wann tritt er auf, wie wird er behandelt, ist es ein Hardware– oder ein reines Software–Ereignis?
7. Der TLB wird mit assoziativen Registern implementiert. Was ist der TLB, was ist ein Register und was ist ein assoziatives Register?
8. Bei einen Seitenfehler wird eine Seite in den Hauptspeicher gebracht und die Seitentabelle modifiziert. Wie wird die Seitentabelle modifiziert? Was ist mit dem TLB, muss er gelöscht werden?
9. Windows–NT und Linux arbeiten mit einer Seitengröße von 4K Byte. Ist das Zufall, ist es ein Fall von einem Plagiat (wer klaute wem die Idee), oder liegt es an der gemeinsamen Hardware? Überlegen Sie: Gibt es eine Hardware–Eigenschaft die mit einer Seitengröße von 4KByte gekoppelt ist?
10. Traditionell ist der logische Adressraum in Segmente (Text, Daten, Stack, Heap, ...) eingeteilt. Kann man die Segmente auf Seiten abbilden, oder ist eine Segmentierung des logischen Adressraums mit Paging nicht vereinbar?
11. Welcher Systemaufruf führt bei einem Unix–System zur Erzeugung eines logischen Adressraums. Was bedeutet die Erzeugung eines neuen Adressraums konkret?

Aufgabe 5, Bonusaufgabe 5, Termin 2. Juli 2002

Nehmen Sie an, Sie sind Mitglied in einer Entwicklergruppe, die Software für einfache eingebettete Systeme entwickelt. Die Zielplattform verfügt über ein Betriebssystem *ohne* virtuellen Speicher. Auf dem System kann nur ein Anwendungsprozess aktiviert werden, dieser kann jedoch beliebig viele Threads starten. Nehmen Sie an, dass für eine bestimmte Klasse von Anwendungen (Threads) viel Speicher in nicht vorhersehbarer Menge benötigt wird (z.B. ein Webserver) – wesentlich mehr als auf dem System zur Verfügung steht. Bei diesen Anwendungen kommt es nicht so sehr auf Geschwindigkeit an. Es ist also möglich Daten auszulagern. Konzipieren Sie eine eine dazu geeignete Software–Infrastruktur, die Threads mit einem jeweils eigenen virtuellen Adressraum ausstattet und diese Adressräume nach dem Konzept *des virtuellen Speichers* mit Hilfe einer Swap–Datei realisiert.

Jeder Thread bekommt seinen eigenen virtuellen Adressraum mit 16–Bit (Wort–) großen Adressen. In diesem Adressraum kann er ohne Beschränkungen agieren. Programmtechnisch definieren Sie dazu eine Basisklasse `VMThread`. Jede Ableitung dieser Klasse hat einen eigenen virtuellen Adressraum von der Größe 2^{16}. Damit werden Programme folgender Art möglich:

```
#include "VMThread.h"
#include <iostream>

class MyThread : public VMThread {
  virtual void run () {
    // gesamten eigenen Adressraum mit '!' fuellen
    // i ist eine logische Adress im Adressraum von this
    //
    for ( LogicalAddress i(this,0x0000); i < 0xffff; ++i )
      *i = '!';

    // Zugriff auf eine logische Adresse 0xef1a
    //
    LogicalAddress adr(this);
    adr = 0xef1a;
    cout << *adr << endl;
  }
};

int main() {
  MyThread t[50]; // 50 Threads mit je 0x10000 Bytes privatem Speicher
  ...
}
```

Ein erster Entwurf für die Klasse `VMThread` ist:

```
typedef ..... Byte;  // ein Byte
typedef ..... Word;  // zwei Bytes

// Schnittstellen zur Implementierung des logischen
// Adressraums
//
class AddressSpaceInt {
 public:
  virtual Byte & getByteAt ( Word address ) = 0;
};

// Basisklasse fuer Threads mit eigenem Adressraum
//
class VMThread : public PThreadNS::PThread {
 public:
  VMThread ();
  virtual ~VMThread () {}

 protected:

  class LogicalAddress {  // logische Adresse adressiert ein Byte
  public:                 // logische Adressen haben Wortgroesse (= 2 Bytes)
    LogicalAddress (VMThread *);
    LogicalAddress (VMThread *, Word);
    bool operator< (const LogicalAddress &) const;
    bool operator> (const LogicalAddress &) const;
    bool operator< (Word) const;
    bool operator> (Word) const;
    LogicalAddress & operator++();
    LogicalAddress & operator= (Word);
    Byte & operator* ();  // Zugriff auf den eigenen Adressraum
```

```
  private:
    Word            w;         // mein Wert
    VMThread &      myThread;  // Zu welchem Thread gehoere ich
  };

  friend class  VMThread::LogicalAddress;

 private:
  AddressSpaceInt & as;   // Implementierung des eigenen Adressraums
};                        // (die Page-Table, Ableitung von AddressSpaceInt)
```

Mit einem ersten Ansatz für die Implementierung:

```
VMThread::VMThread() : as(..???..) {} <--------------

VMThread::LogicalAddress::LogicalAddress(VMThread *vmt)
  : w(0), myThread(*vmt) {}
VMThread::LogicalAddress::LogicalAddress(VMThread *vmt, Word pw)
  : w(pw), myThread(*vmt) {}

bool VMThread::LogicalAddress::operator< (const LogicalAddress &pla) const {
  return w < pla.w;
}
VMThread::LogicalAddress & VMThread::LogicalAddress::operator= (Word adr) {
  w=adr;
  return *this;
}

... etc. ...

Byte & VMThread::LogicalAddress::operator* () {
  return myThread.as.getByteAt (w); <----------------
}
```

Die logischen Adressräume werden in Seiten eingeteilt und im Pagingverfahren bei Bedarf ein und ausgelagert. Dazu muss eine Implementierung des logischen Adressraums `AddressSpace` als Ableitung von `AddressSpaceInt` definiert und `VMThread::as` mit einer Instanz eines solchen Adressraum initialisiert werden.

Ein erster Entwurf für die Klasse `AddressSpace` und des virtuellen Speichers könnte dabei etwa wie folgt aussehen:

```
  class VirtualMemory {
  public:
     ...????....
     PageDescriptor * aquire();
     void release(PageDescriptor * pageTable);
     ...????....
  private:
     static PageDescriptor   pageTables[....]; //Deskriptoren aller Seitentabellen
  };

  class AddressSpace : public AddresSpaceInt {
  public:
    AddressSpace() : pageTable(vm.aquire()) {}  // Seitentabelle erzeugen
    ~AddressSpace () { vm.release( pageTable ); }
```

```
  Byte & getByteAt( Word adr ) {
    // adr zerlegen in SeitenNummer und Offset:
    Byte pageNr     = ...
    Byte pageOffset = ...
    ??... Seite bei Bedarf einlagern ...??
    return .. ?? Adresse pageOffset in der Seite Nr. pageNr ??...
  }

private:
  PageDescriptor *       pageTable; // Zeiger auf Anfang der Seitentabelle fuer dies
  static VirtualMemory & vm;        // Virtueller Speicher in dem
};                                  // der Adressraum realisiert ist


VirtualMemory    virtualMemoryManager;  // der virtuelle Speicher

VirtualMemory &  AddressSpace::vm  = virtualMemoryManager;
```

Definieren Sie einen geeigneten virtuellen Speicher mit FRAME_COUNT Seitenrahmen (z.B. 32) im "Hauptspeicher" und einer Swap–Datei zur Aufnahme der ausgelagerten Seiten. Eine Seite soll mit einem Byte adressierbar sein und ein Byte adressiert jedes Byte innerhalb einer Seite. Ihr Paging–Verfahren soll einen *Second–Chance*–Algorithmus und einem Paging–Dämon einsetzen, der bei einem bestimmten Füllgrad des Hauptspeichers Seiten auslagert.

Der "Hauptspeicher" ist ein statischer Datenbereich im Prozess zu dem die VMThreads gehören. Ebenso gibt es pro Prozess mit VMThreads eine Swap–Datei. Die Seitentabellen werden permanent im Speicher gehalten. Dazu wird ebenfalls wieder im statischen Datenbereich des Prozesses Platz für die Tabellen von MAX_VMTHREADS bereit gehalten. Der Prozess, zu dem VMThreads gehören, übernimmt also für diese die Speicherverwaltung und damit Aktivitäten, die normalerweise dem Betriebssystem zugeordnet werden.