PeANUt Repetition and Virtual Memory

- ref: [PeANUt Spec, sect 3]; additionally [O’H&Bryant, sect 10.1–10.7] or [Null&Lobur, sect 6.5]
- PeANUt repetition
  - macros
  - stack concepts and addressing mode
  - the stack frame and procedures
  - traps
- virtual memory concepts
  - introduction
  - paging
- other issues:
  - MSE marks and feedback (+ some solutions)

PeANUt Repetition – Macros

- context: important (yet simple) computational concept
- widely used in the C programming language
- neither instructions nor procedures! Essentially, just a ‘shorthand’
  - exist only in the assembly language level, i.e. are expanded by the assembler (similarly for C macros)
- ‘regular’ macros
  - are best for ‘straight-line’ code
  - must be defined at the top of the program, used later (e.g. macro.ass)
  - checking/debugging: the .lst file shows expansion of macros
- concise macros are used to give symbolic names to (small) constants (e.g. MAXSIZE = 64) or stack offsets (e.g. x = −3)
  - must be defined (textually) above their first use
  - will override any earlier definition of the same symbolic name
  - e.g. procedure-example1.ass

PeANUt Repetition – The Stack Concepts and Addressing Mode

- a fundamental programming concept! Hence hardware support needed (e.g. SP, !)
- uses a (reserved) part of (normal) memory called the stack item can be efficiently implemented as the memory pointed to by a stack pointer register (SP)
  - stack addressing mode (!): AOP = opspec + SP;
    OP = Memory[AOP]
    - e.g. for load !−3, and if SP=209, AOP=?
  - in diagram, what is value of OP before & after the Push()
- enables return of control to caller (return addresses) & passing parameters and return values

PeANUt Repetition – Procedure Call Convention

- determines order of data in the stack frame
- example C function declaration:
  ```c
  int P(int p1, int p2, ...) {
    int l1, l2, ...;
    ...
  }
  ```
  → 1 RV return value (if any)
  → 2 parameters
  → 3 RA return address
  → 4 local variables (if any)
  → 1 SP just before (explicitly) pushing parameters & just after popping them
  → 2 SP just before executing call P and just after executing P’s ret
  → 3 SP just before executing 1st instruction in P and just before executing P’s ret
  → 4 SP inside body of P, after (explicitly) allocating space for local variables (references to parameters, return value & local variables are relative to this position)
PeANUt Repetition – Procedure Call Convention Examples

```c
void WriteInt(char ch)
{
    printf("\%c", ch);
    ...
    return lx;
}
```

```c
int Log10(int x) {
    return lx;
}
```

### Notes
- The `writeInt` function prints a character to the console.
- The `Log10` function calculates the base 10 logarithm of an integer.

---

PeANUt Repetition – Procedure Call Detail: \( \log = \log_{10}(511) \)

#### Example 1 (PC=1):

- `incsp #1`
- `load #511`
- `incsp #1`
- `store 10`
- `writeInt #1`
- `incsp #1`
- `load 1x`
- `store RV`
- `ret`

#### Example 2 (PC=2):

- `load #511`
- `incsp #1`
- `store 10`
- `writeInt #1`
- `incsp #1`
- `load 1x`
- `store RV`
- `ret`

#### Example 3 (PC=5):

- `call Log10`

---

PeANUt Repetition – Procedure Call Detail: \( \log = \log_{10}(511) \) (II)

### Example 4 (PC=6):

- `load !lx`
- `store !RV`
- `incsp #1`
- `load 10`
- `store log`
- `incsp #1`
- `store 6`
- `ret`

---

PeANUt Repetition – Traps

- **there is a special instruction called trap**
- **used to have PeANUt perform a ‘operating system’ service**
- **depending on the trap’s operand, some particular operation will be performed, e.g.:**
  - trap #1: Halt: tells the PeANUt to stop execution
  - trap #2: Get: Allows you to read a character from keyboard
  - trap #3: Put: Allows you to print out a character
- **some are user-definable/modifiable (lecture P9)**
- **some relate to virtual memory**

Reflection for COMP2300: to PeANUt or not to PeANUt? Possible alternatives:

- **MARIE [Null\&Lobur, Ch 4] – no indexed or stack mode :(**
- **Pep/8 [Computer Systems, J.S. Warford] – much like PeANUt**
- **8088 Assembler/Simulator [Tanenbaum, Appendix C] – subset of thex86 assembly language**
- **a simplified RISC machine?**
Virtual Memory

- **motivation:** (multiple) users regularly need to run jobs whose capacity exceeds that of physical memory (main memory)
  - one of the first (and most important) instances of virtualization
  - also, in a multiprocessing environment, each program 'sees' memory in the same way, regardless of where it is executing in physical memory
- **virtual memory** is a technique whereby program-addressable memory is made to appear to be larger than physical memory
- needed because there is a memory hierarchy:
  - many different mediums for the storage of data
  - generally, there is a *trade-off* between speed and capacity (fast memories tend to be small; large memories tend to be slow)
- virtual memory is a technique whereby program-addressable memory is made to appear to be larger than physical memory
- needed because there is a memory hierarchy:
  - many different mediums for the storage of data
  - generally, there is a *trade-off* between speed and capacity (fast memories tend to be small; large memories tend to be slow)

<table>
<thead>
<tr>
<th>medium</th>
<th>access time (nsec)</th>
<th>typical size</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>~10</td>
<td>&lt; 1 KB</td>
</tr>
<tr>
<td>cache memory</td>
<td>~25</td>
<td>&lt; 2 MB</td>
</tr>
<tr>
<td>physical memory</td>
<td>100</td>
<td>&lt; 2 GB</td>
</tr>
<tr>
<td>disk</td>
<td>20,000,000</td>
<td>&gt; 10 GB</td>
</tr>
</tbody>
</table>

[O'H&Bryant, fig 6.21]

Virtual Memory – Nomenclature

- **address space:** range of addresses accessible to programmer
- **logical/virtual addresses:** addresses as seen by the programmer
- **physical addresses:** actual addresses in main memory
- **the Memory Management Unit (MMU)** performs this translation

Paging

- how do we share main memory between competing chunks of the (virtual memory) address space?
  - one solution is called paging
  - **break all memory into equal sized chunks called pages**
  - when accessing a virtual address, check if the corresponding page is in main memory
    - (if not, move it into main memory and then access it)
- exploits locality of (address) references:
  - memory accesses tend not to be random (in a program, they are often in a sequence)
  - consider in particular accesses involved with instruction fetching
  - rule of thumb: a program spends about 90% of its time in only 10% of the code

Paging Issues

- how big should a page be? Influenced by disk technology
  - needs to be large enough to amortize costs of overheads (disk block seek time; also page book-keeping costs)
  - but if too large, causes fragmentation
- what does main memory look like? It consists of a mixed group of pages, with each page occupying a slot (page frame)
  - (e.g. PeANUI VM)
- what does (disk) virtual memory look like? It consists of all of the pages
- programmer's view of paging:
  - is oblivious of it: all program addresses are virtual
  - can only see performance degradation (when paging requires many disk accesses, called swapping)
Virtual Memory Issues

● what pages should be resident in main memory (MM) at any time?
  ■ the most used pages (in the near future)
  ■ different paging policies give an approximation to ‘most used’

● data consistency:
  ■ upon a page fault (access of data in a page not currently in physical memory),
    a page currently in main memory usually has to be removed:
    ◆ if simply thrown out, data may be lost
    ◆ if always written back to disk (upon each store), will be too slow!
  ■ solution: write it back to disk, if it has been written to (made dirty)
    ◆ hence the MMU must record this for each page (‘Dirty bit’)