Nov 6 and 8, 2006
Lecture 11: Memory Management

September 5, 2007

1 ADMIN

Please start to read Chapter 9 on Memory Management.

We finish up some material from last lecture – the Banker's algorithm for deadlock-free resource allocation.

2 REVIEW

- Q: Describe the Banker's algorithm (only one resource, money).

  A: For any request, if allocating it is safe, do it. Safety: if there is a way to deplete the currently allocations to 0. While there is a non-zero allocation A_i to some process p_i, if the current free amount f is at least as great as the claimed amount c_i (or, need amount) we let $f = f + A_i$ and set $A_i = 0$. When all $A_i = 0$, we are safe. Else, not.

- Q: Describe the four conditions that are necessary and sufficient for a deadlock.

  A: (1) Exclusive use of resource (2) Hold and wait policy (3) Non-preemption (4) Circular wait

- Q: Suppose we have three programs, prog1.c, prog2.c, prog3.c. Only prog1.c has the main routine defined, but it calls routines defined in prog2.c and prog3.c. How do you compile such a collection of programs?

  A:

- Q: How do you write the program prog1.c, prog2.c, prog3.c?

  A: Nothing to be done for prog2.c and prog3.c. But in prog1.c, if you refer to a routine (say, gcd(a,b)) defined in another file, you must use the extern declaration:

  extern int gcd (int, int);

3 Address Binding

- Between compiling and running of a computer program (as a process), we have several stages of transformations.
- A very interesting entity to observe in this transformation is the concept of an ADDRESS of an object.

FILE	TRANSFORMATION	ADDRESS
c program		symbolic address (names)
↓	compile	
*.s file		symbolic relocatable address
↓	assemble	
*.o file		binary relocatable address
↓	link	
*.exe file		logical address
↓	load	
core image		physical address

- One way to view the address transformation is that more and more details are fixed (or "bound").

4 Separate Compilation

- Separate compilation of C programs.

1. Suppose that we have the following program (prog1.c) that uses a gcd(a,b) routine:

```
///////////////////////////////////////////////
// file: prog1.c
//    This program takes a sequence of 2 or more
//    positive integers and returns their common gcd.
///////////////////////////////////////////////

#include <stdio.h>

extern int gcd(int, int);

int main(int argc, char** argv) {

   if (argc<=2) {
       printf("Need two or more integers!\n");
       return 0;
   }

   int m = atoi(argv[1]);
   int n = atoi(argv[2]);
   int g = gcd(m,n);
   int i;

   for (i=3; i<argc; i++){
       m=g;
       n=atoi(argv[i]);
       g=gcd(m,n);
   }
   printf("Gcd = %d\n", g);
   return 0;

}//main
///////////////////////////////////////////////
```

2.
```
////////////////////////////////////
// file: prog2.c
```

```
//     This has the gcd routine.
//     It has no main routine!
////////////////////////////////////

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

int gcd(int m, int n){
   int tmp;
   if (m<n) swap(&m,&n);
   while (n>0) {
      tmp = n;
      n= m%n;
      m=tmp;
   }
   return m;
}
////////////////////////////////////
```

3. To compile them, we would execute these two commands:

   `% gcc -c prog2`

   The output is an object file called file prog2.o. We now link this to the main program in prog1.c:

   `% gcc prog1.c prog2.o`

   The output is an executable file `a.exe` (or `a.out`). Now you can run the program:

   ```
   % a.out 18 30 15 123

   Gcd = 3
   ```

- We have learned that we can suppress linking by the `-b` flag:

  `% gcc -c prog.c`

  suppresses the linking, producing an object file `prog.o`. This means that we have (1) compiled the program and (2) assembled the program into a relocatable (binary) program.

- In fact, we can even suppress the assembly process with the `-S` flag:

  `% gcc -S prog.c`

  just compiles and produces the symbolic file `prog.s` (which is an ascii file that you can read with an ordinary editor). Try this!

  This is what I get on my solaris:

```
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        .file "prog2.c"
        .section ".text"
        .align 4
        .global swap
        .type swap, #function
        .prog 020
swap:
        save %sp, -120, %sp
        st %i0, [%fp+68]
        st %i1, [%fp+72]
```

```
        ld [%fp+68], %g1
        ld [%g1], %g1
        st %g1, [%fp-20]
        ...
        st %g1, [%g2]
        restore
        jmp %o7+8
        nop
        .size swap, .-swap
        .global .rem
        .align 4
        .global gcd
        .type gcd, #function
        .proc 04
gcd:
        save %sp, -120, %sp
        st %i0, [%fp+68]
        st %i1, [%fp+72]
        ...
        nop
.LL7:
        ld [%fp+72], %g1
        ...
.LL6:
        ld [%fp+72], %g1
        ...
        jmp %o7+8
        nop
        .size gcd, .-gcd
        .ident "GCC: (GNU) 4.0.2"
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

5 Using Dynamic Library

- What is dynamic library?

  Normally, linking creates a copy of the library routine and puts them into your executable file.

  Shared library just puts a STUB there.

  At run time, this STUB will know how to call and load the routine (and if it is already loaded, we just directly call it). Thus the loaded routine can be shared by other processes.

  YOU NEED OS SUPPORT FOR THIS!

- Suppose you want to create a dynamic library containing the routines of prog2.c:

  `% gcc -fPIC -c prog2.c -o prog2.o`

  `% gcc -shared prog2.o -o libprog2.so`

  NOTE: PIC stands for "position-independent code".

- Now you can link prog1.c to this dynamic library:

  ```
  % gcc prog1.c -L. -lprog2 -o prog1
  % prog1 18 30 15 123
  Gcd = 3
  ```

- What have we gained? The dynamically linked program `prog1` should be smaller in size than the statically linked `a.out`: For our specific programs here, we can check:

```
% wc prog1 a.out
68   331   7028  prog1
45   403   7416  a.out
```

  The numbers represents lines, words and bytes in the respective files (words are sequences of characters separated by white spaces). Since these are binary files, line count and word count are meaningless. We see that `prog1` is indeed a smaller file than the statically linked `a.out`.

- To see that `prog1` is dynamically linked, you can use a UNIX tool called ldd+ (list dynamic dependencies). On my solaris sparc station:

```
% ldd prog1
libprog2.so =>   ./libprog2.so
libc.so.1 =>     /usr/lib/libc.so.1
libgcc_s.so.1 =>         /usr/local/lib/libgcc_s.so.1
libdl.so.1 =>    /usr/lib/libdl.so.1
/usr/platform/SUNW,Sun-Blade-1000/lib/libc_psr.so.1
```

6 Process Swapping

- MEMORY MODEL: Main Memory (or RAM) and Secondary Memory (or Disk).
- Processes must be in Main Memory to run.
- Backing Store: reserved space on Disk for processes not currently active.
- SWAPPING: the movement of processes between Main Memory and Backing Store.
- Typical time to swap between Main Memory and Backing Store: 416 ms.

  (See calculation in text: 1 MB image at 5MB/sec transfer rate takes 200 ms. Add some overhead)

  This is expensive, so must use it carefully!
- On UNIX: swapping is only enabled with many processes are running the Main Memory usage has exceeded some threshhold.

7 Memory Allocation

- Simplest model: consecutive space is allocated for each process image.
- This creates HOLES. Holes that are too small for allocation is wasted. This is the FRAGMENTATION problem. (To be more precise, EXTERNAL fragmentation)
- Implementation:

  We need to maintain two data structures called the **FREE MEMORY** and **USED MEMORY**.

  We need to implement two functions: **free memory** and **allocate memory**.
- Free Memory:

  Typically a linked list of all the contiguous blocks of available memory. These are in sorted order to allow merging. Similarly for Used Memory.
- Free Memory Function:

  Add the returned block to the free list Merge adjacent free blocks if possible.

- Allocate Memory Function:

  Find a free block that is large enough, and allocate part of the block. Retain the rest in free memory.

- STRATEGIES TO ALLOCATE SPACE:

  1. First Fit (OK).
     ACTUALLY, there is an implicit notion that "first" means as it occurs in the free memory list. But one can generalize this to mean free in some other criteria. In this case, this can even give us "best fit".
  2. Best Fit or "Smallest fit" (OK).
     The smallest one that fits. To implement this, we need another data structure to to list the free blocks in size order.
  3. Worst Fit (NO)
     Idea: avoid getting small useless blocks. PROBLEM: you quickly deplete your biggest blocks, and this may prevent future allocation for a large request.
  4. Quick Fit
     Maintain a separate pool of typical block sizes...

8 Relocatable Addressing

- CPU generates a LOGICAL ADDRESS (0 to some limit)
- This is transformed into a PHYSICAL ADDRESS.
- Hardware and RUNTIME support for this: MMU has a RELOCATION REGISTER holding a value.
- PHYSICAL ADDRESS = LOGICAL ADDRESS + RELOCATION REGISTER
- REMARK: sometimes "LOGICAL" is called "VIRTUAL". E.g., "virtual address" and "virtual memory".
- Definition: RELOCATION is the translation of logical address to physical address.
- Applications: simple memory protection, and allows compaction of processes in main memory.

9 Paging

- Paging is a solution to the problem of (external) fragmentation and removes the need for contiguous memory allocation.
- All modern memory allocation use some form of this scheme.
- Needed: concept of VIRTUAL PAGES (or LOGICAL PAGES) and FRAMES. Pages and frames are two sides of the coin: one in logical space, other is physical space.
- Need a PAGE TABLE to map from pages to frames.
- E.g., Suppose we have 32 bit address space, and each address refers to a byte. This is 4 GB. Suppose each page is 4 KB. This means we have 1 Million pages. If each page table is 4 bytes, we need 4 MB of memory for the page table.
- So one problem is large page table size (each process needs one of these tables!)
- Solution 1: put page table in main memory. This means that each memory reference requires at least 2 references to main memory.

  Then each process only need to keep ONE register ("Page Table Base Register" (PTBR)) to point to the location in main memory.

- Solution 2:

  Use a specialized hardware cache called TRANSLATION LOOK-ASIDE BUFFER (TLB). This is an associative memory: each entry is a pair (key,value). Typically, TLB has 64 to 1024 entries.

  HOW to use TLB? It contains only some entries from the page table which is store in Main Memory (as in Solution 1).

  In case of a TLB MISS, we act as in Solution 1. But we also add this new entry to TLB. Thus, we need some cache replacement policy.

  NOTE: "Wired down entries" means they will never be replaced. Useful for critical pages.

- REFINEMENT to Solution 2:

  We need to refresh TLB with each context switch.

  One way to avoid this is to store with each TLB Entry a identifier (ASID's) for the process. This way, we do not need to flush the TLB with each context switch!