CS252: Systems Programming

Gustavo Rodriguez-Rivera

Computer Science Department
Purdue University

General Information

- Web Page: http://www.cs.purdue.edu/homes/cs252
- Office: LWSN1210
- E-mail: grr@cs.purdue.edu
- Textbook:
  - No textbook. We will use my notes and selected material in the web.
  - We have a draft of an online book authored by Gustavo Rodriguez-Rivera and Justin Ennen available in the website.
- Recommended:
  - Advanced Programming in the UNIX Environment by W. Richard Stevens. (Useful for the shell. Good as a reference book.)

Mailing List

- All announcements will be sent via email.
- Mailing List will be created automatically

PSOs

- There is lab the first week.
- The projects will be explained in the labs.
- E-mail questions to
  cs252-ta@cs.purdue.edu
- TAs office hours will be posted in the web page.

Grading

- Grade allocation
  - Midterm: 25%
  - Final: 25%
  - Projects and Homework: 40%
  - Attendance 10%
- Exams also include questions about the projects.

Course Organization

1. Address space. Structure of a Program. Text, Data, BSS, Stack Segments.
2. Review of Pointers, double pointers, pointers to functions
3. Use of an IDE and debugger to program in C and C++.
4. Executable File Formats. ELF, COFF, a.out.

Course Organization

5. Development Cycle, Compiling, Assembling, Linking. Static Libraries

6.Loading a program, Runtime Linker, Shared Libraries.

7. Scripting Languages. sh, bash, basic UNIX commands.

8. File creation, read, write, close, file mode, IO redirection, pipes, Fork, wait, waitpid, signals, Directories, creating, directory list

Course Organization

9. Project: Writing your own shell.
10. Programming with Threads, thread creation.
11. Race Conditions, Mutex locks.
12. Socket Programming. Iterative and concurrent servers.
13. Memory allocation. Problems with memory allocation. Memory Leaks, Premature Frees, Memory Smashing, Double Frees.

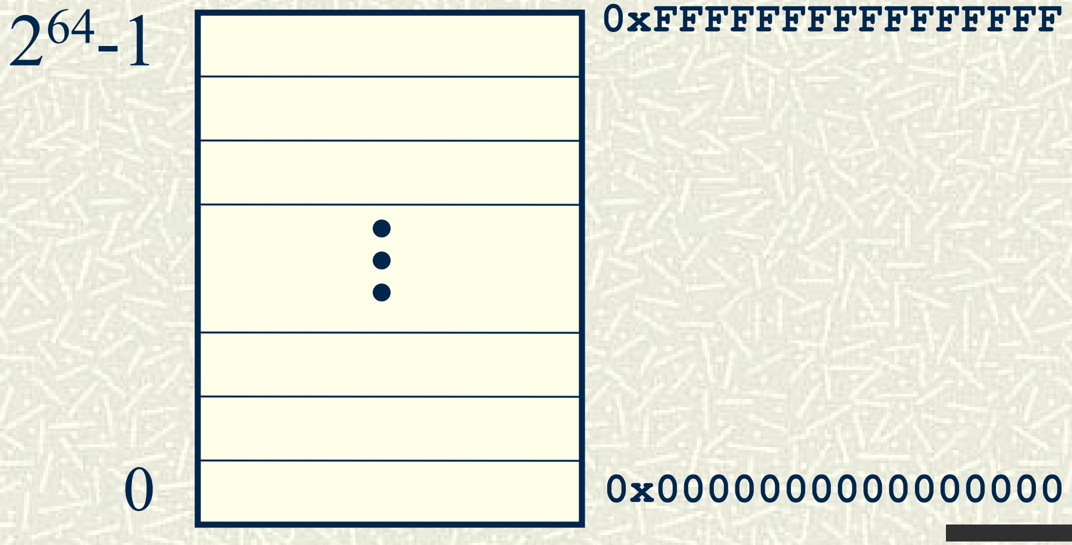
Course Organization

14. Introduction to SQL
15. Source Control Systems (CVS, SVN) and distributed (GIT, Mercurial)
16. Introduction to Software Engineering
17. Design Patterns
18. Execution Profiling.

Program Structure

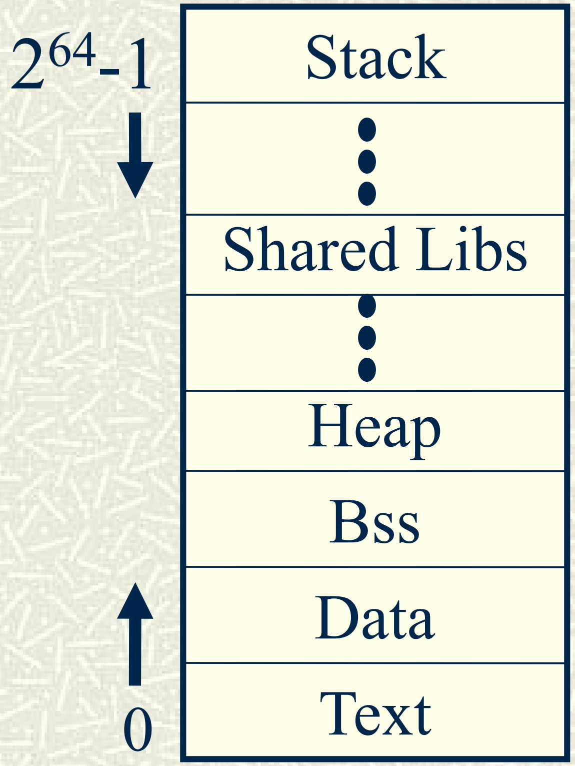
Memory of a Program

- A program sees memory as an array of bytes that goes from address 0 to 2^{64}-1 (`0x0000000000000000`-`0xFFFFFFFFFFFFFFFF`)
- That is assuming a 64-bit architecture.



Memory Sections

- The memory is organized into sections called “memory mappings”.



Memory Sections

- Each section has different permissions: read/write/execute or a combination of them.
- Text - Instructions that the program runs
- Data – Initialized global variables.
- Bss – Uninitialized global variables. They are initialized to zeroes.
- Heap – Memory returned when calling malloc/new. It grows upwards.
- Stack – It stores local variables and return addresses. It grows downwards.

Memory Sections

- Dynamic libraries – They are libraries shared with other processes.
- Each dynamic library has its own text, data, and bss.
- Each program has its own view of the memory that is independent of each other.
- This view is called the "Address Space" of the program.
- If a process modifies a byte in its own address space, it will not modify the address space of another process.

Example

```
Program hello.c
int a = 5;    // Stored in data
  section
int b[20];    // Stored in bss
int main() { // Stored in text
  int x;      // Stored in stack
  int *p =(int*)
     malloc(sizeof(int)); //In heap
}
```

Memory Gaps

- Between each memory section there may be gaps that do not have any memory mapping.
- If the program tries to access a memory gap, the OS will send a SEGV signal that by default kills the program and dumps a core file.
- The core file contains the value of the variables global and local at the time of the SEGV.
- The core file can be used for "post mortem" debugging.

```
gdb program-name core
gdb> where
```

What is a program?

- A program is a file in a special format that contains all the necessary information to load an application into memory and make it run.
- A program file includes:
  - machine instructions
  - initialized data
  - List of library dependencies
  - List of memory sections that the program will use
  - List of undefined values in the executable that will be known until the program is loaded into memory.

Executable File Formats

- There are different executable file formats
  - ELF – Executable Link File

    It is used in most UNIX systems (Solaris, Linux)
  - COFF – Common Object File Format

    It is used in Windows systems
  - a.out – Used in BSD (Berkeley Standard Distribution) and early UNIX

    It was very restrictive. It is not used anymore.
- Note: BSD UNIX and AT&T UNIX are the predecessors of the modern UNIX flavors like Solaris and Linux.

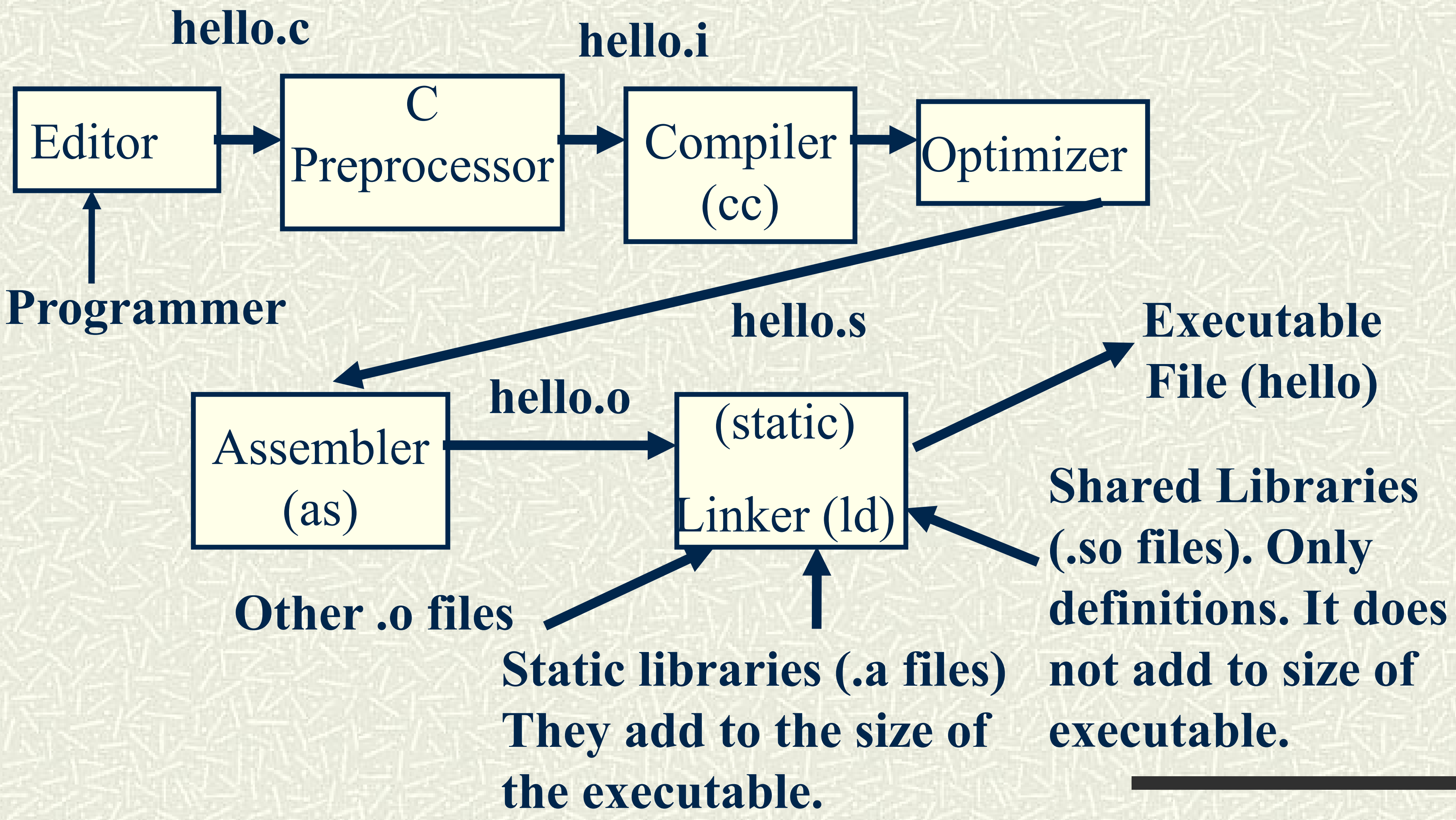
Building a Program

- The programmer writes a program hello.c
- The ***preprocessor*** expands #define, #include, #ifdef etc preprocessor statements and generates a hello.i file.
- The ***compiler*** compiles hello.i, optimizes it and generates an assembly instruction listing hello.s
- The ***assembler*** (as) assembles hello.s and generates an object file hello.o
- The compiler (cc or gcc) by default hides all these intermediate steps. You can use compiler options to run each step independently.

Building a program

- The linker puts together all object files as well as the object files in static libraries.
- The linker also takes the definitions in shared libraries and verifies that the symbols (functions and variables) needed by the program are completely satisfied.
- If there is symbol that is not defined in either the executable or shared libraries, the linker will give an error.
- Static libraries (.a files) are added to the executable. shared libraries (.so files) are not added to the executable file.

Building a Program


hello.c
hello.i
Editor
C Preprocessor
Compiler (cc)
Optimizer
Programmer
hello.s
Executable File (hello)
hello.o
Assembler (as)
(static) Linker (ld)
Shared Libraries (.so files). Only definitions. It does not add to size of executable.
Other .o files
Static libraries (.a files) They add to the size of the executable.

Original file hello.c

```
#include <stdio.h>
main()
{
   printf("Hello\n");
}
```

After preprocessor

gcc -E hello.c > hello.i

(-E stops compiler after running preprocessor)

hello.i:

```
/* Expanded /usr/include/stdio.h */
 typedef void *__va_list;
 typedef struct __FILE   __FILE;
 typedef int     ssize_t;
 struct FILE {…};
 extern int fprintf(FILE *, const char *, ...);
 extern int fscanf(FILE *, const char *, ...);
 extern int printf(const char *, ...);
 /* and more */
 main()
 {
    printf("Hello\n");
 }
```

After assembler

gcc -S hello.c   (-S stops compiler after assembling)

hello.s:

```
        .align 8
.LLC0:  .asciz   "Hello\n"
.section         ".text"
          .align 4
          .global main
          .type    main,#function
          .proc    04
main:     save     %sp, -112, %sp
          sethi    %hi(.LLC0), %o1
          or       %o1, %lo(.LLC0), %o0
          call     printf, 0
          nop
.LL2:     ret
```

After compiling

- "gcc -c hello.c" generates hello.o
- hello.o has undefined symbols, like the ***printf*** function call that we don't know where it is placed.
- The main function already has a value relative to the object file hello.o

csh> nm -xv hello.o

```
hello.o:
[Index]    Value      Size      Type  Bind  Other Shndx   Name
[1]     |0x00000000|0x00000000|FILE |LOCL |0    |ABS    |hello.c
[2]     |0x00000000|0x00000000|NOTY |LOCL |0    |2      |gcc2_compiled
[3]     |0x00000000|0x00000000|SECT |LOCL |0    |2      |
[4]     |0x00000000|0x00000000|SECT |LOCL |0    |3      |
[5]     |0x00000000|0x00000000|NOTY |GLOB |0    |UNDEF  |printf
[6]     |0x00000000|0x0000001c|FUNC |GLOB |0    |2      |main
```

After linking

- “`gcc -o hello hello.c`” generates the hello executable
- Printf does not have a value yet until the program is loaded

```
csh> nm hello
[Index]    Value         Size         Type  Bind  Other Shndx    Name
[29]     |0x00010000|0x00000000|OBJT |LOCL |0     |1        |_START_
[65]     |0x0001042c|0x00000074|FUNC |GLOB |0     |9        |_start
[43]     |0x00010564|0x00000000|FUNC |LOCL |0     |9        |fini_dummy
[60]     |0x000105c4|0x0000001c|FUNC |GLOB |0     |9        |main
[71]     |0x000206d8|0x00000000|FUNC |GLOB |0     |UNDEF    |atexit
[72]     |0x000206f0|0x00000000|FUNC |GLOB |0     |UNDEF    |_exit
[67]     |0x00020714|0x00000000|FUNC |GLOB |0     |UNDEF    |printf
```

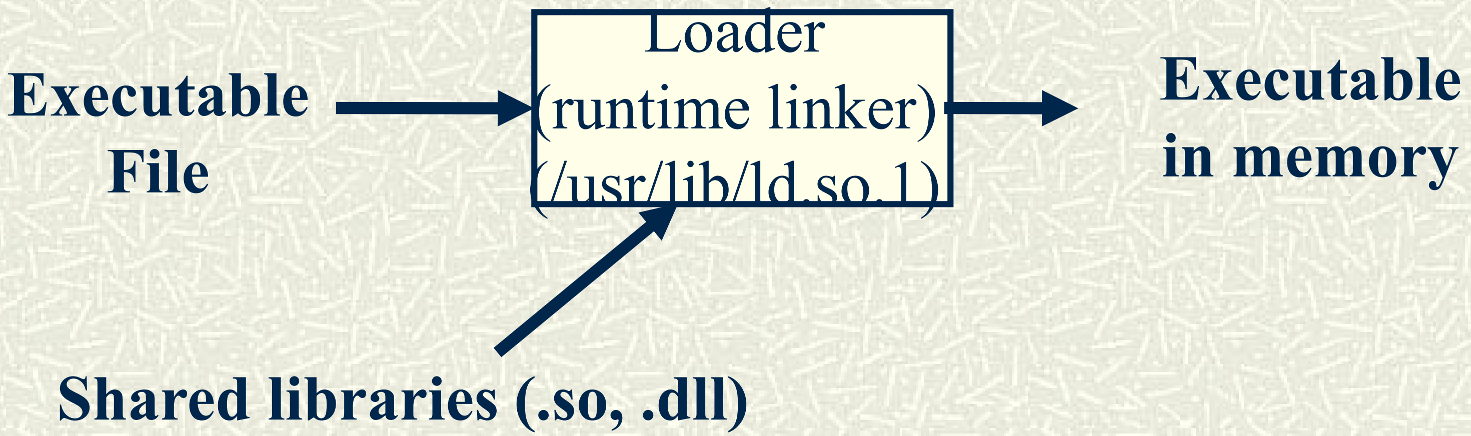
Loading a Program

- The loader is a program that is used to run an executable file in a process.
- Before the program starts running, the loader allocates space for all the sections of the executable file (text, data, bss etc)
- It loads into memory the executable and shared libraries (if not loaded yet)

Loading a Program

- It also writes (resolves) any values in the executable to point to the functions/variables in the shared libraries.(E.g. calls to printf in hello.c)
- Once memory image is ready, the loader jumps to the _***start*** entry point that calls init() of all libraries and initializes static constructors. Then it calls ***main()*** and the program begins.
- _***start*** also calls ***exit()*** when ***main()*** returns.
- The loader is also called “runtime linker”.

Loading a Program


Executable File
Loader (runtime linker) (/usr/lib/ld.so.1)
Executable in memory
Shared libraries (.so, .dll)

Static and Shared Libraries

- Shared libraries are shared across different processes.
- There is only one instance of each shared library for the entire system.
- Static libraries are not shared.
- There is an instance of an static library for each process.

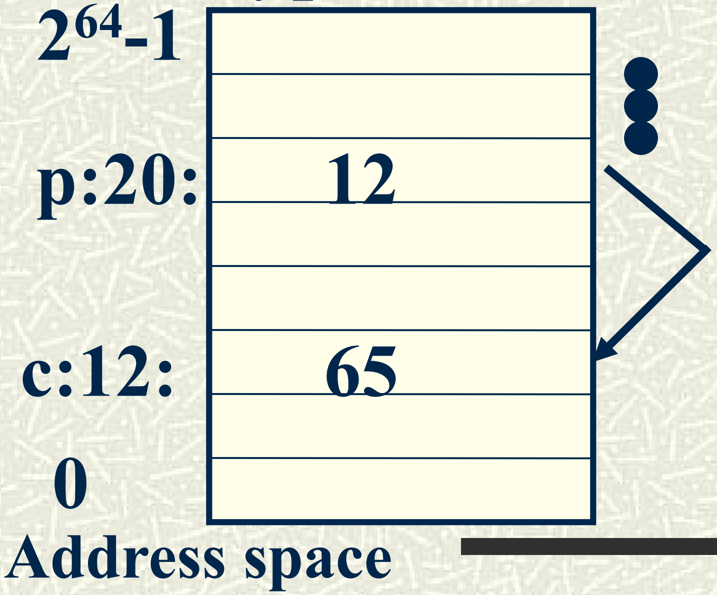
Static and Dynamic

- Static – Events that happen during program building.
  - Example: Static linker
- Dynamic – Events that happen while program is running.
  - Also called “Runtime”.
  - Example: Dynamic linker

Memory and Pointers

- A pointer is a variable that contains an address in memory.
- In a 64 bit architectures, the size of a pointer is 8 bytes independent on the type of the pointer.

```
Char c = 'A';  //ascii 65

char * p =  &c;
```



Ways to get a pointer value

1. Assign a numerical value into a pointer

   Char * p = (char *) 0x1800;

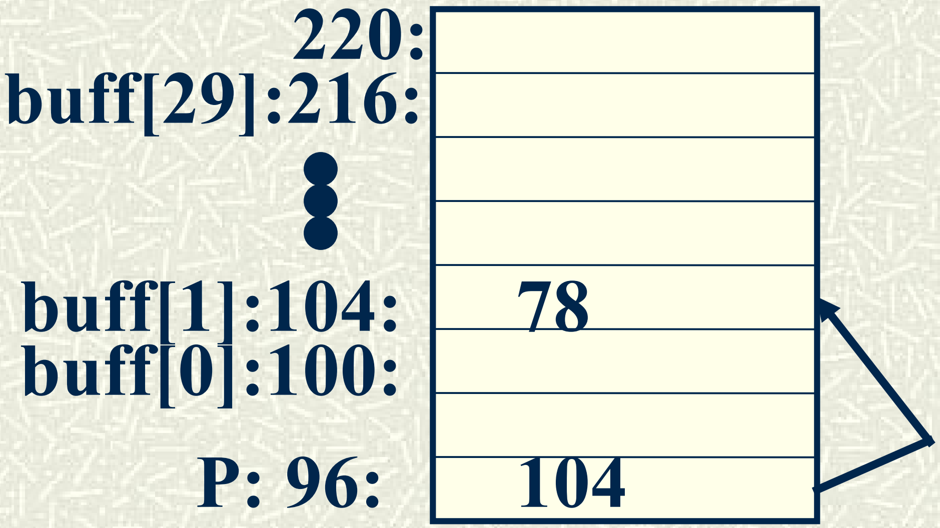
   *p = 5; // Store a 5 in location 0x1800;

   Note: Assigning a numerical value to a pointer isn't recommended and only left to programmers of OS, kernels, or device drivers

Ways to get a pointer value

2. Get memory address from another variable:

```
int *p;
int buff[ 30];
p = &buff[1];
*p =78;
```



Ways to get a pointer value

3. Allocate memory from the heap

```
int *p
p = new int;
int *q;
q = (int*)malloc(sizeof(int))
```

Ways to get a pointer value

- You can pass a pointer as a parameter to a function if the function will modify the content of the parameters

```
void swap (int *a, int *b){
        int temp;
        temp=*a;
        *a=*b;
        *b=temp;
}
In main: swap(&x, &y)
```

Common Problems with Pointers

- When using pointers make sure the pointer is pointing to valid memory before assigning or getting any value from the location
- String functions do not allocate memory for you:
  char *s;
  strcpy(s, "hello"); --> SEGV(uninitialized pointer)
- The only string function that allocates memory is strdup (it calls malloc of the length of the string and copies it)

Printing Pointers

- **It is useful to print pointers for debugging**

```
char*i;
char buff[10];
printf("ptr=%d\n", &buff[5])
```

Or In hexadecimal

```
printf("ptr=0x%x\n", &buff[5])
```

Instead of using printf, I recommend to use **`fprintf(stderr, ...)`** since stderr is unbuffered and it is guaranteed to be printed on the screen.

`sizeof()` operator in Pointers

- The size of a pointer is always 4 bytes in a 32 bit architecture independent of the type of the pointer:

```
sizeof(int)==4 bytes
sizeof(char)==1 byte
sizeof(int*)==4 bytes
sizeof(char*)==4 bytes
```

Using Pointers to Optimize Execution

- Assume the following function that adds the sum of integers in an array using array indexing.

```
int sum(int * array, int n)
{
   int s=0;
   for(int i=0; i<n; i++)
   {
    s+=array[i]; // Equivalent to

   //*(int*)((char*)array+i*sizeof(int))
   }
   return s;
```

Using Pointers to Optimize Execution

- Now the equivalent code using pointers

```
int sum(int* array, int n)
{
   int s=0;
   int *p=&array[0];
   int *pend=&array[n];
   while (p < pend)
   {
    s+=*p;
    p++;
   }
   return s;
}
```

Using Pointers to Optimize Execution

- When you increment a pointer to integer it will be incremented by 4 units because sizeof(int)==4.
- Using pointers is more efficient because no indexing is required and indexing require multiplication.
- Note: An optimizer may substitute the multiplication by a “<<“ operator if the size is a power of two. However, the array entries may not be a power of 2 and integer multiplication may be needed.

Array Operator Equivalence

- We have the following equivalences:

```
int a[20];
a[i]          - is equivalent to
*(a+i)        - is equivalent to
*(&a[0]+i)  - is equivalent to
*((int*)((char*)&a[0]+i*sizeof(int)))
```

- You may substitute array indexing `a[i]` by `*((int*)((char*)&a[0]+i*sizeof(int)))` and it will work!
- ***C was designed to be machine independent assembler***

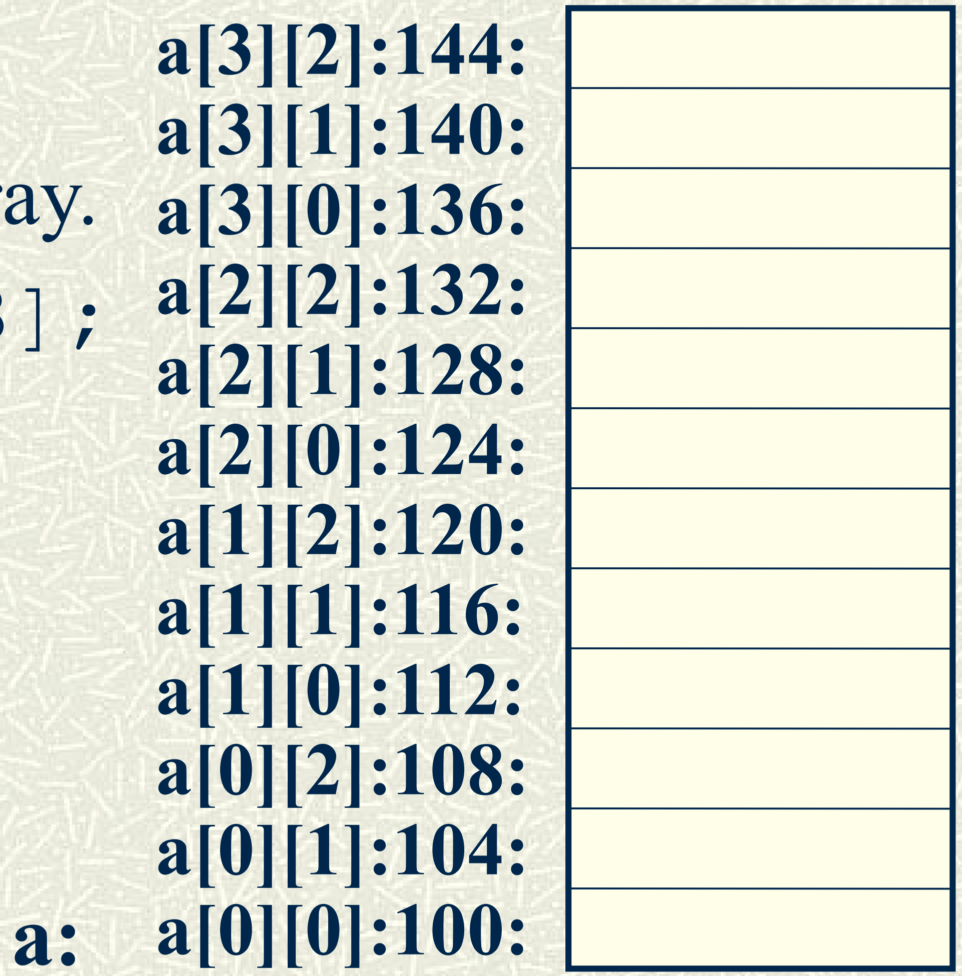
2D Array. 1st Implementation

- 1st approach

Normal 2D array.

```
int a[4][3];
```

```
a[i][j] ==
*(int*)((char*)a +
i*3*sizeof(int) +
j*sizeof(int))
```



2D Array 2nd Implementation (Jagged Arrays)

- 2nd approach

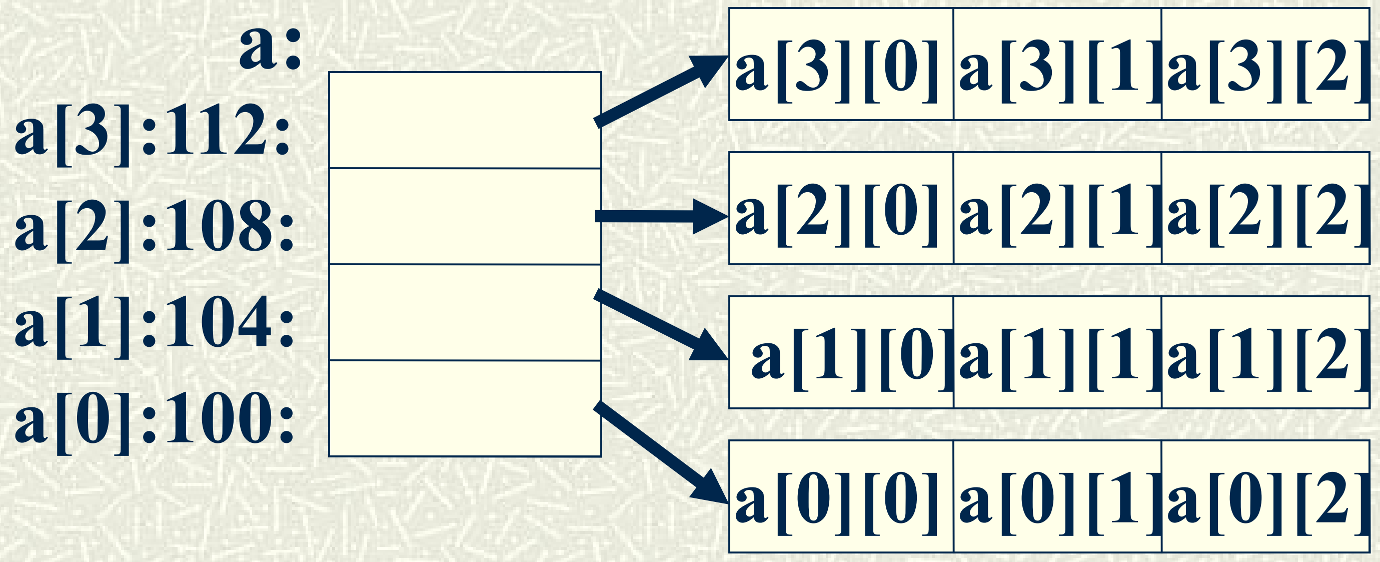
Array of pointers to rows

```
int*(a[4]);
for(int i=0; i<4; i++){
      a[i]=(int*)malloc(sizeof(int)*3);
      assert(a[i]!=NULL);
}
```

2D Array 2nd Implementation

- 2nd approach

Array of pointers to rows (cont)



```
int*(a[4]);

a[3][2]=5
```

2D Array 3rd Implementation (Jagged Arrays)

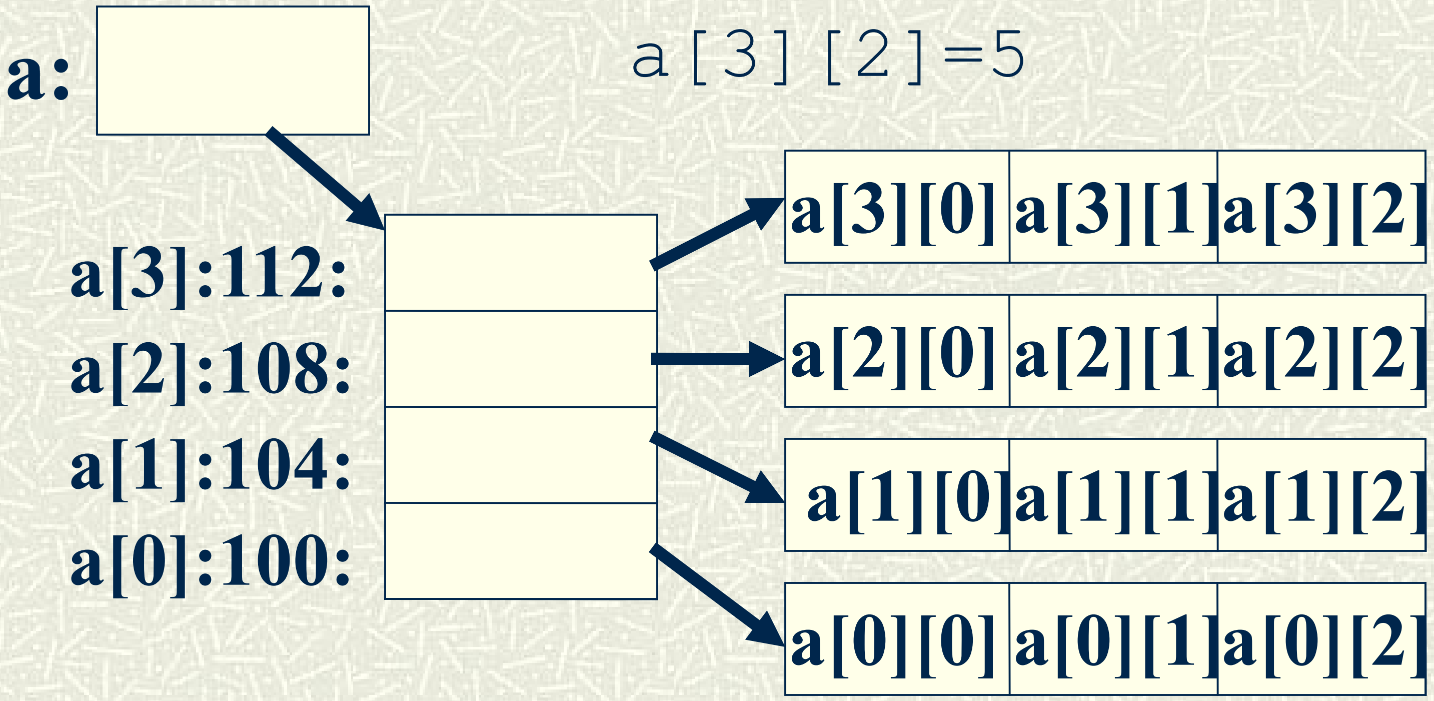
- 3rd approach. a is a pointer to an array of pointers to rows.

```
int **a;
a=(int**)malloc(4*sizeof(int*));
assert( a!= NULL)
for(int i=0; i<4; i++)
{
  a[i]=(int*)malloc(3*sizeof(int));
  assert(a[i] != NULL)
}
```

2D Array 3rd Implementation

- a is a pointer to an array of pointers to rows. (cont.)

```
int **a;
a[3][2]=5
```

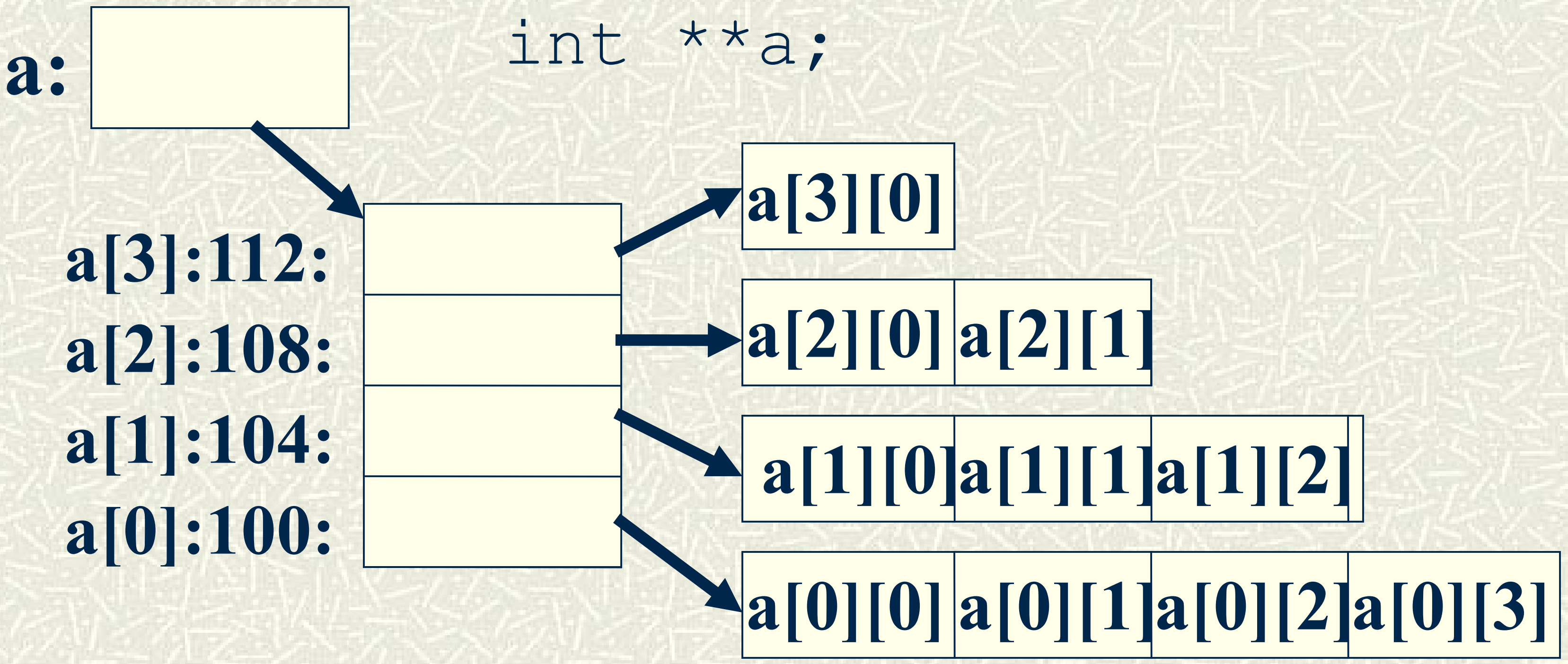


Advantages of Pointer Based Arrays

- You don’t need to know in advance the size of the array (dynamic memory allocation)
- You can define an array with different row sizes

Advantages of Pointer Based Arrays

- Example: Triangular matrix



Pointers to Functions

- Pointers to functions are often used to implement Polymorphism in “C”.
- ***Polymorphism***: Being able to use the same function with arguments of different types.
- Example of function pointer:

```
 typedef void (*FuncPtr)(int a);
```

- `FuncPtr` is a type of a pointer to a function that takes an “`int`” as an argument and returns “`void`”.

An Array Mapper

```
typedef void (*FuncPtr)(int a);

void intArrayMapper( int *array, int n, FuncPtr func ) {
 for( int = 0; i < n; i++ ) {
    (*func)( array[ i ] );
    }
}
int s = 0;
void sumInt( int val ){
    s += val;
}
void printInt( int val ) {
    printf("val = %d \n", val);
}
```

Using the Array Mapper

```
int a[ ] = {3,4,7,8};
main( ){
   // Print the values in the array
   intArrayMapper(a, sizeof(a)/sizeof(int), printInt);

   // Print the sum of the elements in the array
   s = 0;
   intArrayMapper(a, sizeof(a)/sizeof(int), sumInt);
   printf("total=%d\", s);
}
```

A More Generic Mapper

```
typedef void (*GenFuncPtr)(void * a);
void genericArrayMapper( void *array,
      int n, int entrySize, GenFuncPtr fun )
{
  for( int i = 0; i < n; i++; ){
     void *entry = (void*)(
        (char*)array + i*entrySize );
      (*fun)(entry);
  }
}
```

Using the Generic Mapper

```
void sumIntGen( void *pVal ){
   //pVal is pointing to an int
   //Get the int val
   int *pInt = (int*)pVal;
   s += *pInt;
}

void printIntGen( void *pVal ){
   int *pInt = (int*)pVal;
   printf("Val = %d \n", *pInt);
}
```

Using the Generic Mapper

```
int a[ ] = {3,4,7,8};
main( ) {
// Print integer values
   s = 0;
  genericArrayMapper( a, sizeof(a)/sizeof(int),
            sizeof(int), printIntGen);

   // Compute sum the integer values
   genericArrayMapper( a, sizeof(a)/sizeof(int),
           sizeof(int), sumIntGen);
   printf("s=%d\n", s);

}
```

Swapping two Memory Ranges

- In the lab1 you will implement a sort function that will sort any kind of array.
- Use the array mapper as model.
- When swapping two entries of the array, you will have pointers to the elements (`void *a, *b`) and the size of the entry `entrySize.`

```
void * tmp = (void *) malloc(entrySize);
assert(tmp != NULL);
memcpy(tmp, a, entrySize);
memcpy(a,b , entrySize);
memcpy(b,tmp , entrySize);
```

- Note: You may allocate memory only once for tmp in the sort method and use it for all the sorting to save muliple calls to malloc. Free tmp at the end.

Generic Sorting Esqueleton

```
 mysort(void * array, int ascending, int n, int elemSize, ComFunc comp)
//Assume bubblesort
for(int j=0; ….) {
  for( int i=0; ….) {
     // Get address of item i
     char * p = ((char*) array+i*elemSize);
     //Get address of iten i+1
    char *q = ((char*) array+(i+1)*elemSize);
   int result= comp((void*)p,(void*)q;
  //Swap if necessary
}
```

String Comparison in Sort Function

- In lab1, in your sort function, when sorting strings, you will be sorting an array of pointers, that is, of "char* entries.
- The comparison function will be receiving a “pointer to char*” or a” char**” as argument.

```
int StrComFun( void *pa, void *pb) {
  char** stra  =  (char**)pa;
  char ** strb  = (char**)pb;
  return strcmp( *stra, *strb);
}
```

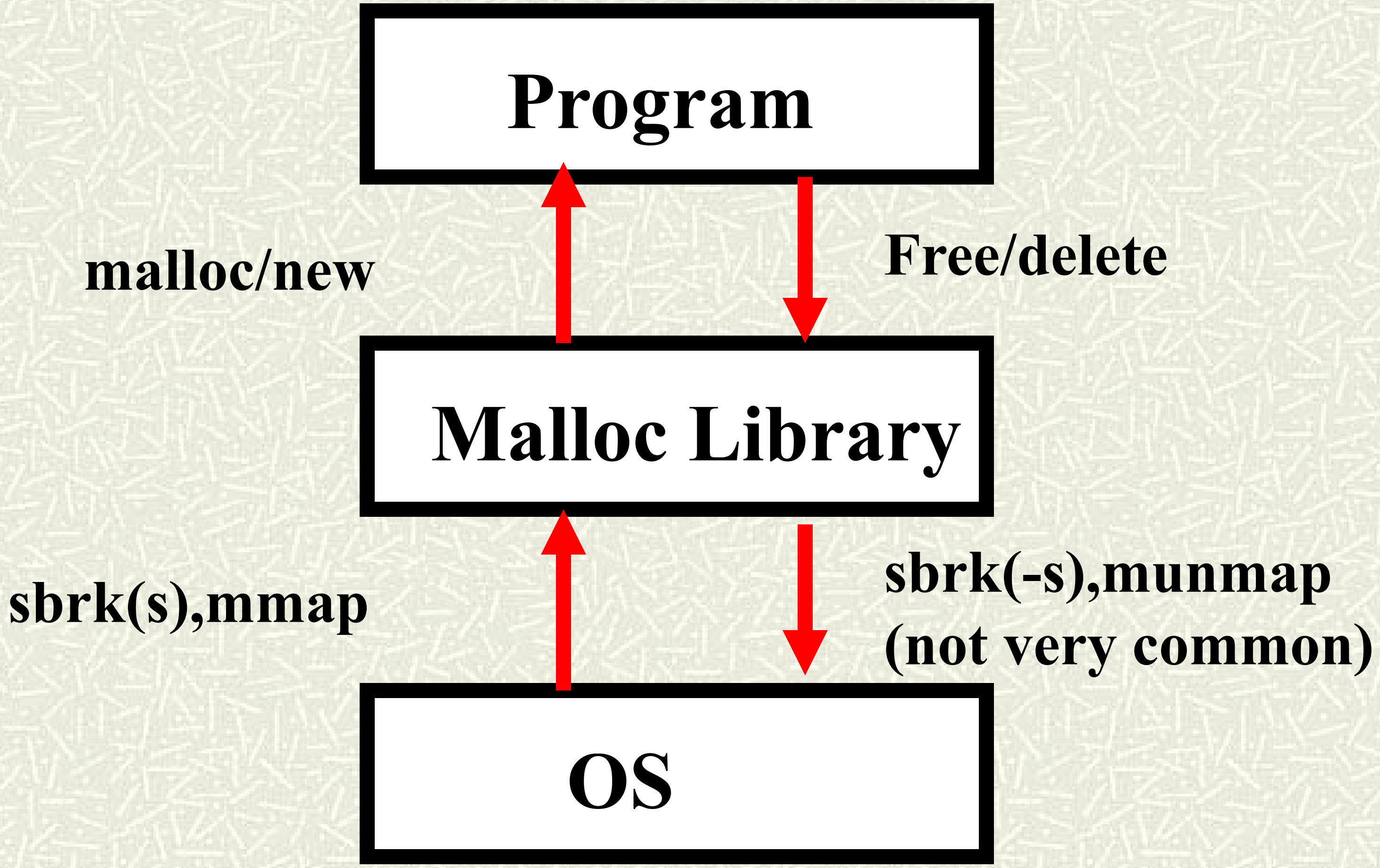
Explicit Memory Management

- We need to call malloc to get memory at runtime.
- In explicit memory management a program frees objects by calling free/delete explicitly
- The program uses a “malloc library” usually provided by the C standard library libc.
- Memory is requested from the OS and added to a free list.
- Subsequent requests are satisfied from memory from the free list.
- Free/delete returns memory to the free list.

Explicit Memory Management

- Memory is requested to the OS in big chunks (E.g. 8KB).
- This decreases the number of times that memory is requested from the OS
- Also, it is difficult to return memory to the OS since the OS handles memory in pages and memory allocator handle memory in bytes.

Explicit Memory Management



Explicit Memory Management

When the memory allocator runs out of memory in the free list it calls sbrk(s).

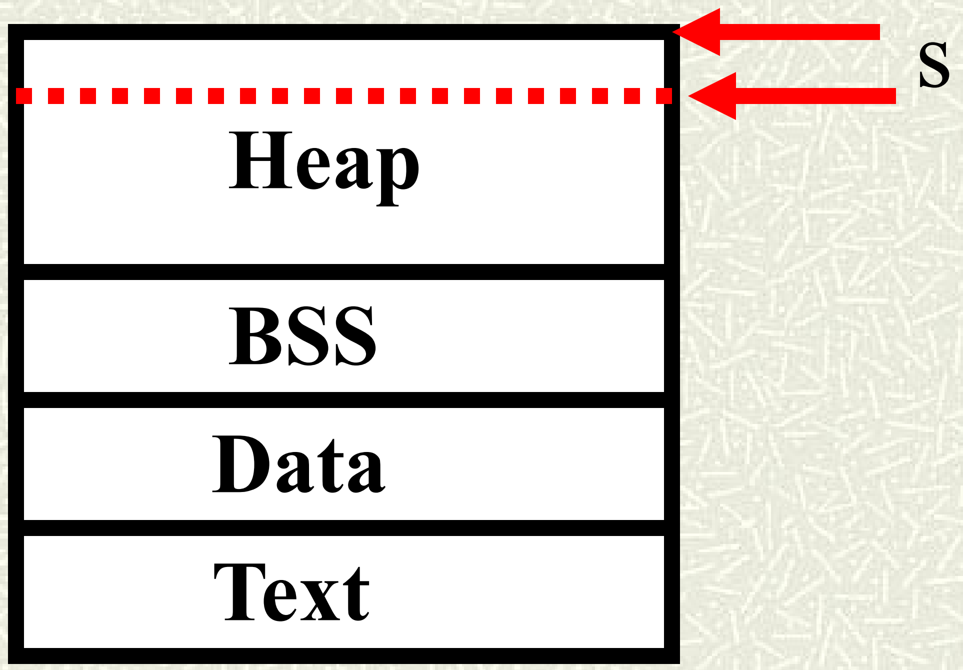
Sbrk(s) increases the size of the heap by s bytes.

Sbrk retuns a pointer to the old heap limit.

You can decrease the size of the heap by passing a negative size to sbrk. This will shrink the heap.

Explicit Memory Management

sbrk(s)



Malloc Implementation

There are several data structures that can be used for memory allocation:

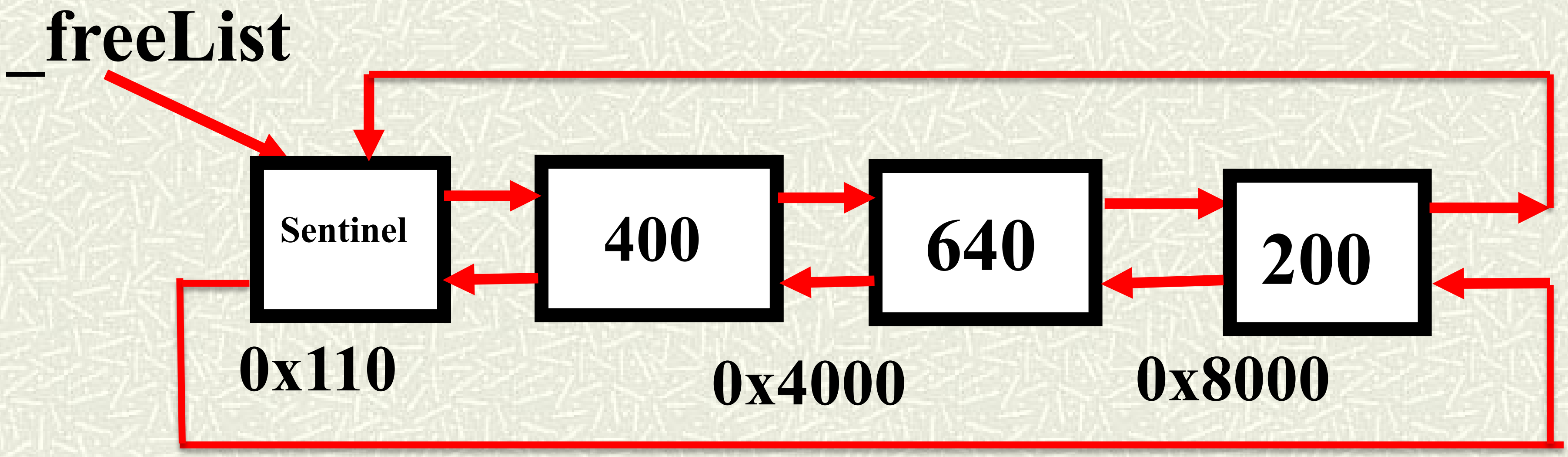
- Single Free list
- Segregated Free Lists
- Cartesian Tree
- Boundary tags

Malloc Single Free List

In this implementation of malloc all the free memory is stored in a single free list.

Each chunk in the free list has a header with the size of the chunk and a pointer to the next chunk.

Malloc Single Free List



Malloc Single Free List

- During allocation a chunk of the right size if searched, split if necessary and the remainder is returned to the free list.
- When an object is freed, the free list is traversed to find the right place of insertion and the object is coalesced if possible.
- If the request cannot be satisfied with the existing free list, the allocator gets memory from the OS.
- The request for memory to the OS is larger (E.g. 8KB - 2MB) than the size requested by malloc. This reduces the number of times memory is requested from the OS since this is slow.
- Chunks in the list may be ordered by address.
  - Allocation: O(n) where n = # of chunks
  - Free: O(n)

Malloc Single Free List

- Two Allocation Policies:
  - First Fit:
    - Use the first chunk in the free list that satisfies the request.
  - BestFit
    - Choose smallest chunk that satisfies the request.

First Fit vs. Best Fit

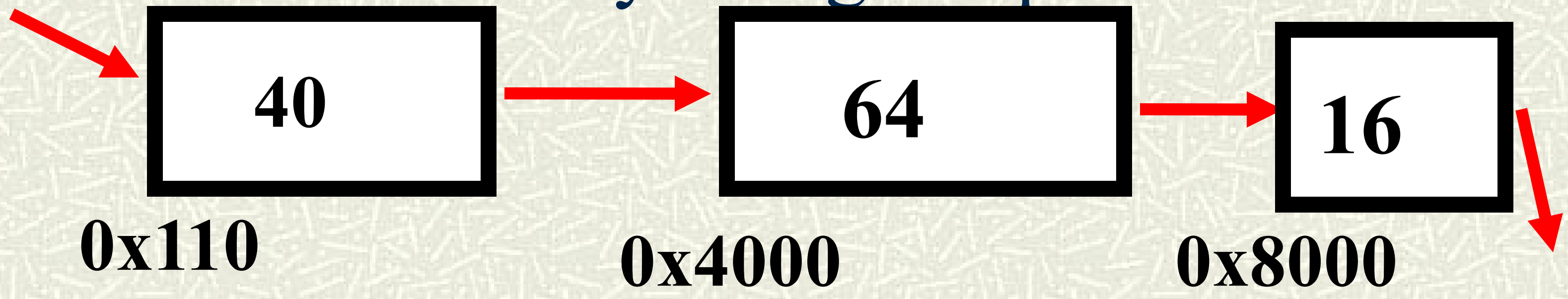
- First Fit is faster
- Best Fit uses less space.
- In first fit if the search always starts at the beginning, it will create many small objects at the beginning of the list.
- Next fit:
  - It is like first fit but the next search starts where the last ended. This speeds first fit and it will prevent the accumulations of small chunks at the beginning of the list.

Notes on malloc

- Malloc also allocates memory for a header. The header stores the size of the object.
- The header will be needed when the object is freed.
- Malloc will return a pointer after the header.
- The memory returned by malloc is aligned to 8 bytes, that is, the address of the object is a multiple of 8 bytes.
- This is because RISC architectures need certain types to be stored in aligned memory. Doubles have to be stored in 8 byte boundaries in memory.
- If you try to store an int value into a non-aligned address in SPARC you will get a SIGBUS error. In the x86 architecture you will not get an error but the execution will slow down.

External Fragmentation

External Fragmentation is the waste of memory in the free list due to small non-contiguous blocks that cannot satisfy a large request.



p = malloc(100);

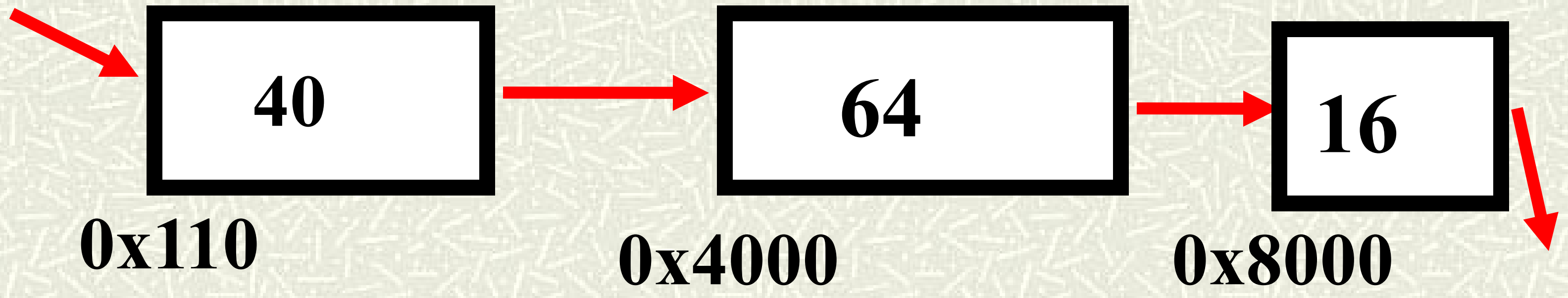
100 bytes object + 8 bytes header = 108 bytes total

The allocation cannot be satisfied even though the free list has more than 108 bytes. The allocator will need to get more memory from the OS.

External Fragmentation

The external fragmentation can be measured as:

Ext. Fragmentation%=100*(1-size_largest_block/total_mem_in_free_list)



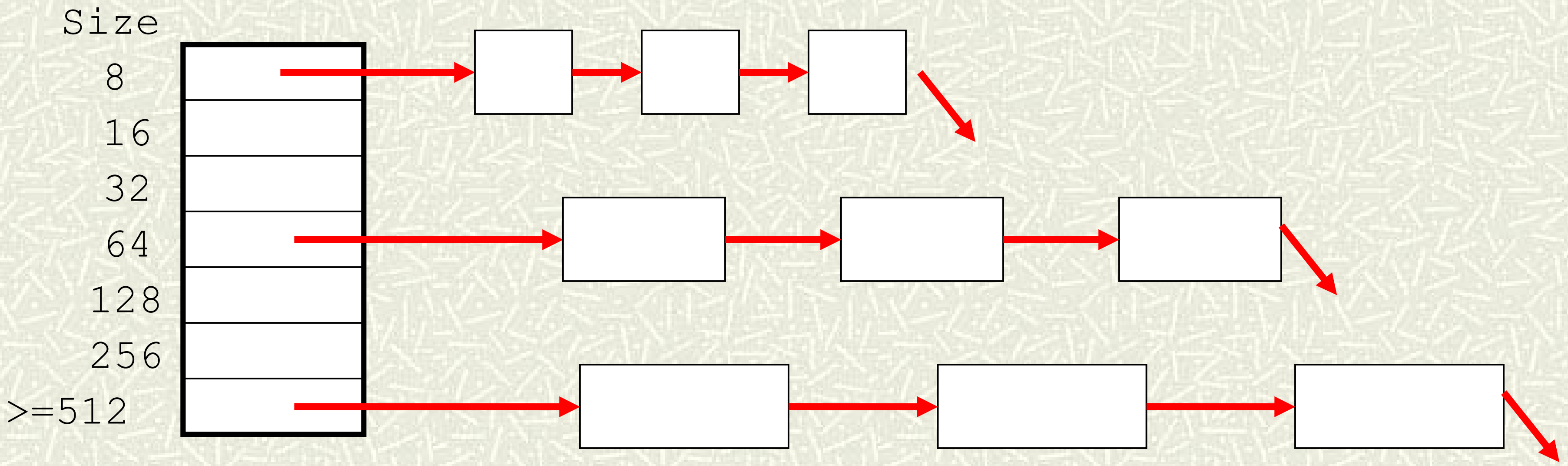
Ext. Fragmentation%=100(1-64/120)=47%

If there is only one block in the list ext fragmentation=0%

Segregated Free Lists

It is another data structure for memory allocation.

There are multiple free lists for different sizes.



Segregated Free Lists

- Very often the fixed sizes are powers of two.
- Objects are allocated from the free list of the nearest size.
- If a free list is empty, the allocator gets a page of memory from the OS and populates the corresponding fre list.
- Some implementations of free lists do not have coalescing. Once an object is created for a size it will be of that size for the entire execution of the program.

Segregated Free Lists

- Coalescing an object would require traversing all the blocks to find out if the neighboring blocks are free
- With segregated free lists:
  - Allocation (smalll objects)= O(1)
  - Free (small objects) =O(1) no coalescing.
- Segregated free list allocators are fast but they use more memory.
- The BSD UNIX allocator uses segregated free lists

External and Internal Fragmentation

External Fragmentation:

Waste of memory due to having non-contiguous blocks in the free list that cannot satisfy an allocation for a large object.

Internal Fragmentation:

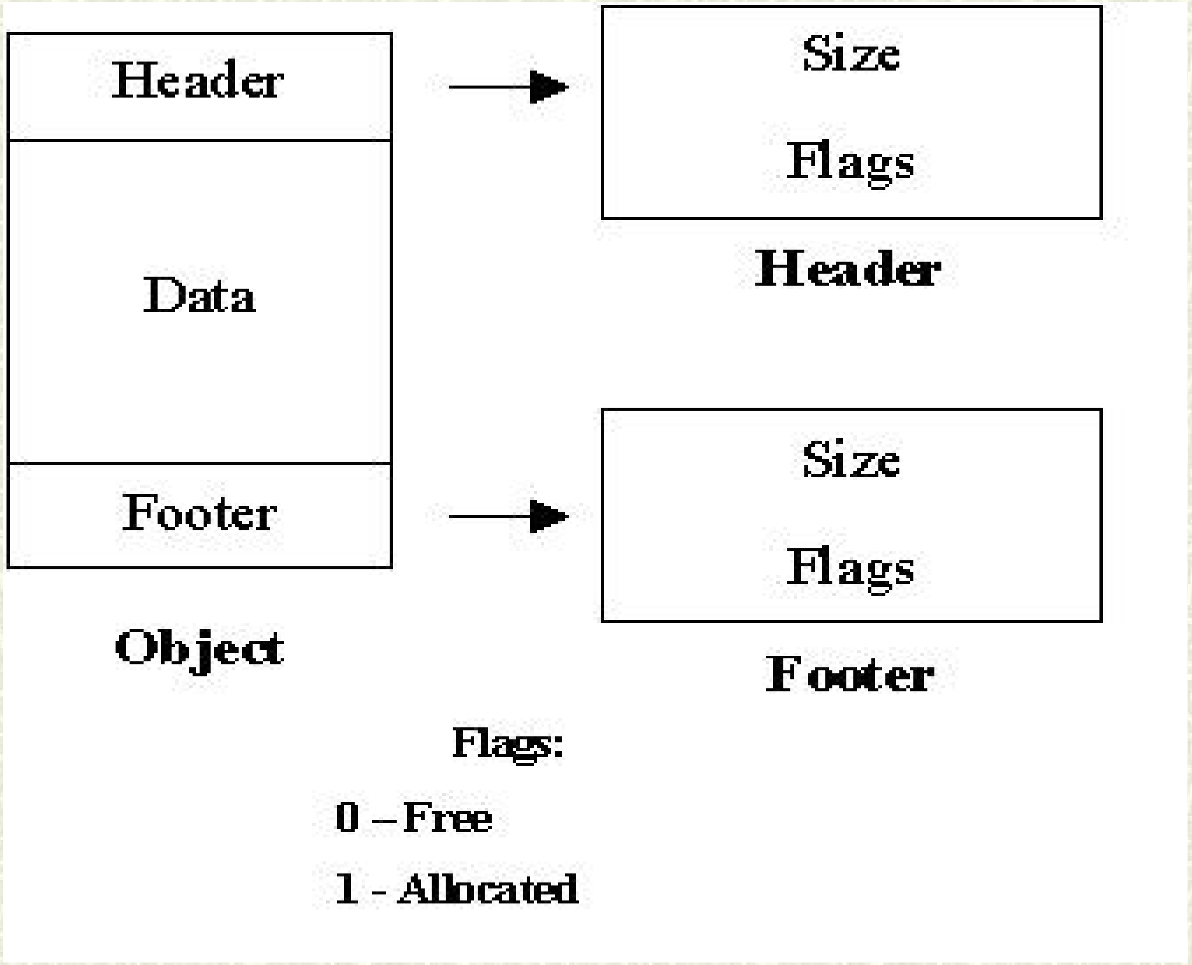
Waste of memory due to the allocator returning a larger block than the memory requested.

Internal Frag = 100*(1 – Size_requested/Mem Allocated)

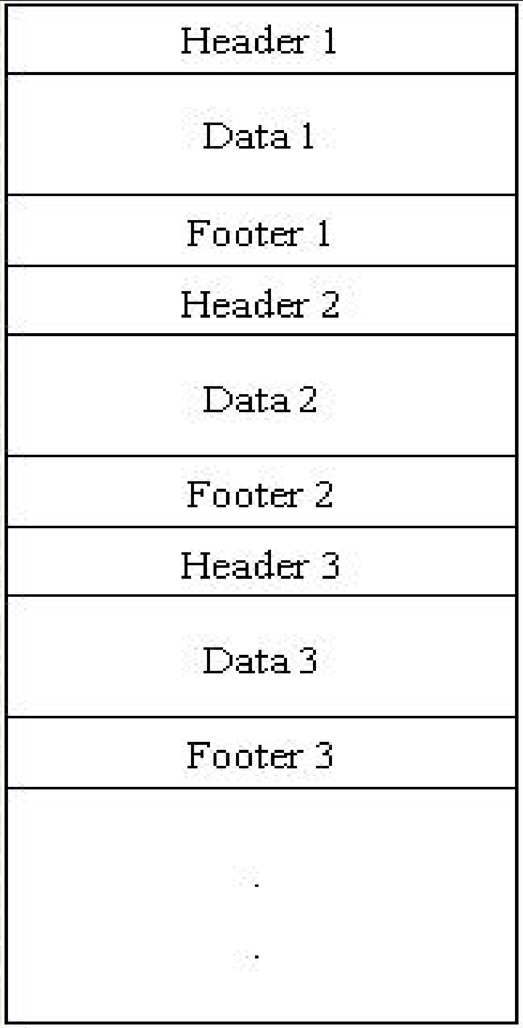
Boundary Tags

- Boundary Tags allow identifying the neighboring objects in O(1). This allows coalescing of objects in O(1)
- Each object ***allocated or free*** has a header and a footer.
- The header and the footer contain the size of the object as well as a flag that tells if the object is allocated or not.
- When freeing an object it is possible to check if the neighbors are free in constant time and coalesce them if possible.

Boundary Tags



Boundary Tags

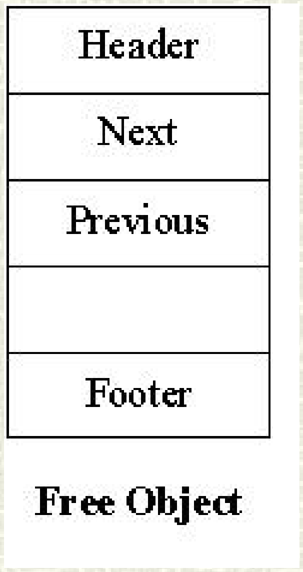


Boundary Tags

The memory allocator you will implement objects in one single free list.

Boundary Tags

- When the object is free, the header also contains a pointer to the next object in the list and a pointer to the previous object in the list.
- This allows removal of an object in the middle of a list from a list in constant time. That is needed after coalescing.



Allocation Algorithm

1. Round up requested size to the next 8 byte boundary (if size ==0 assume 1 byte data)
2. Add the size of the header and the footer:
   - real_size = roundup8(requested size) + sizeof(header) + sizeof(footer)

Allocation Algorithm

- 3. Search for the first object that is large enough to satisfy the request and split it. Use one portion of the object to satisfy the allocation request and return the remainder to the corresponding list.
- 4. If the remainder is smaller than the size of the header plus the footer plus next plus previous making the object unusable, then do not split the object and use the entire object to satisfy the allocation.

Allocation Algorithm

5. If there is not enough memory in the free lists to satisfy the allocation, then request memory from the OS using sbrk(). Round up the request of memory to the OS to a 16KB boundary.

Free Algorithm

1. Check the footer of the left neighbor object (the object just before the object being freed) to see if it is also free. If that is the case, remove the left neighbor from its free list using the previous and next pointers and coalesce this object with the object being freed.

Free Algorithm

2. Check the header of the right neighbor object (the object just after the object being freed) to see if it is also free. If that is the case, remove the right neighbor from its free list using the previous and next pointers and coalesce it with the object being freed.
3. Place the freed object in the corresponding free list and update the header and footer.

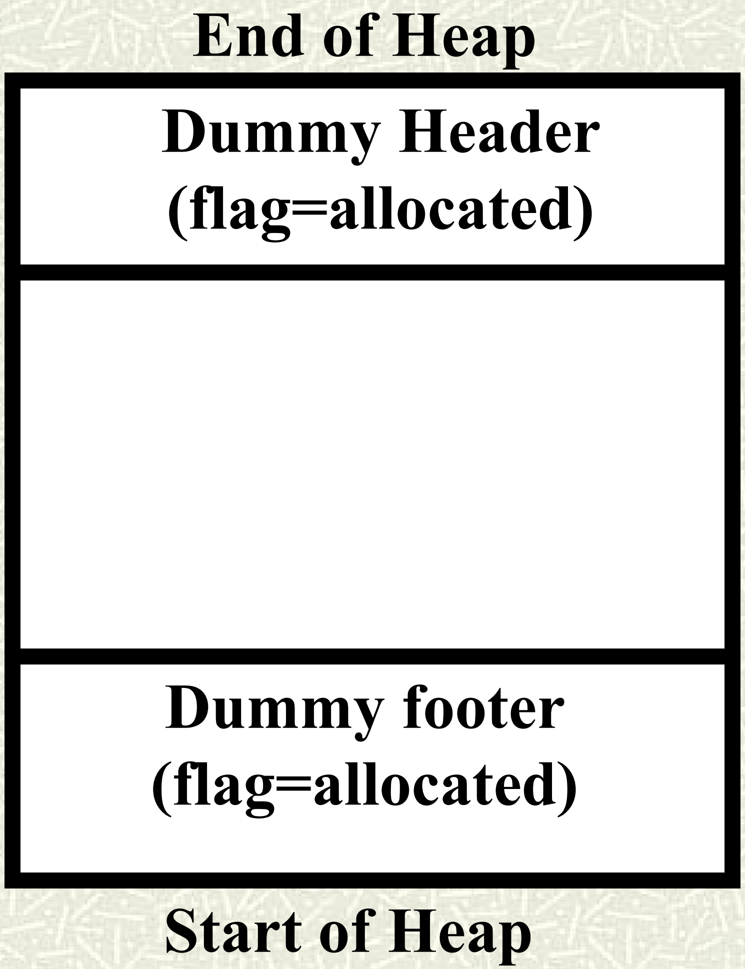
Fence Posts

- If the object freed is at the beginning of the heap, it is likely that your allocator will try erroneously to coalesce memory beyond the beginning of the heap.
- Also other libraries may call sbrk() causing a hole in the heap.

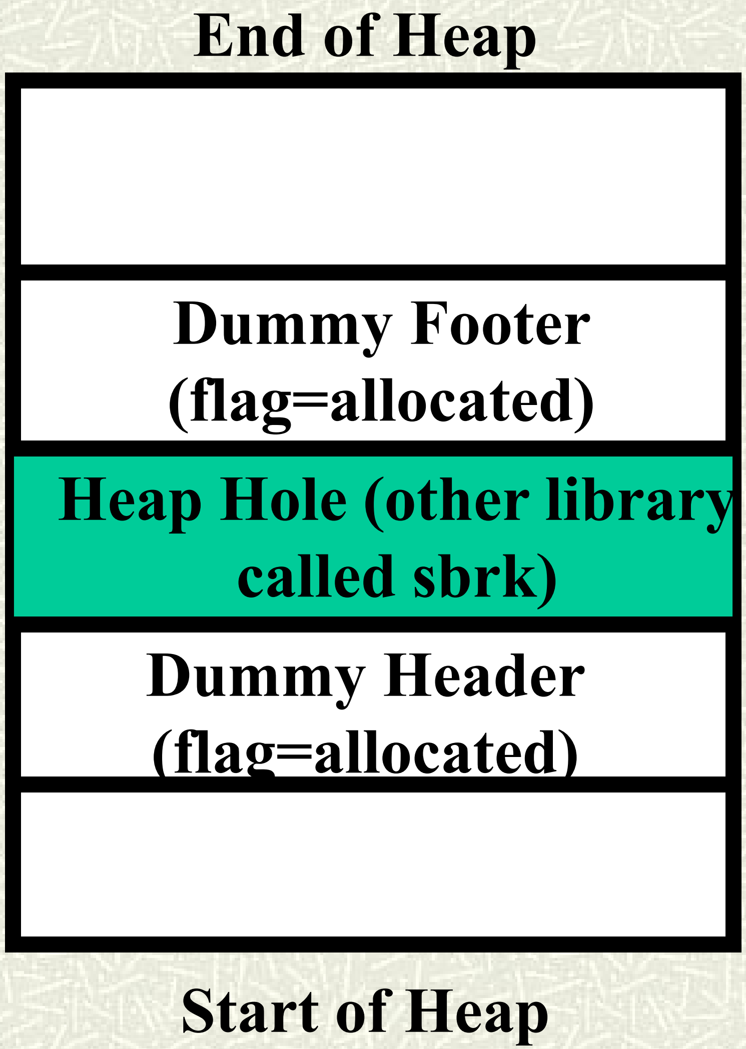
Fence Posts

- To prevent coalescing with memory beyond the beginning or the end of the chunks that belong to the heap your allocator will:
  - every time a chunk of memory is requested from the OS, your allocator has to add a dummy footer at the beginning of the chunk with the flag set to “allocated”.
  - Also at the end of the chunk you will add a header with the flag set to “allocated”.
- If two chunks are consecutive, you should remove the fence posts between them.

Fence Posts


End of Heap
Dummy Header
(flag=allocated)
Dummy footer
(flag=allocated)
Start of Heap

Fence Posts


End of Heap
Dummy Footer (flag=allocated)
Heap Hole (other library called sbrk)
Dummy Header (flag=allocated)
Start of Heap

DL Malloc

- The standard memory allocator used in Linux systems is Doug Lea’s malloc
- This implementation is public domain:

  http://g.oswego.edu/dl/html/malloc.html

Memory Allocation Errors

- Explicit Memory Allocation (calling free) uses less memory and is faster than Implicit Memory Allocation (GC)
- However, Explicit Memory Allocation is Error Prone

1. Memory Leaks
2. Premature Free
3. Double Free
4. Wild Frees
5. Memory Smashing

Memory Leaks

- Memory leaks are objects in memory that are no longer in use by the program but that ***are not freed***.
- This causes the application to use excessive amount of heap until it runs out of physical memory and the application starts to swap slowing down the system.
- If the problem continues, the system may run out of swap space.
- Often server programs (24/7) need to be “rebounced” (shutdown and restarted) because they become so slow due to memory leaks.

Memory Leaks

- Memory leaks is a problem for long lived applications (24/7).
- Short lived applications may suffer memory leaks but that is not a problem since memory is freed when the program goes away.
- Memory leaks is a “slow but persistent disease”. There are other more serious problems in memory allocation like premature frees.

Memory Leaks

Example:

```
while (1) {
  ptr = malloc(100);
}
```

Premature Frees

- A premature free is caused when an object that is still in use by the program is freed.
- The freed object is added to the free list modifying the next/previous pointer.
- If the object is modified, the next and previous pointers may be overwritten, causing further calls to malloc/free to crash.
- Premature frees are difficult to debug because the crash may happen far away from the source of the error.

Premature Frees

Example:

```
int * p = (int *)malloc(sizeof(int));
* p = 8;
free(p); // free adds object to free list
         // updating next and prev
…
*p = 9;  // next ptr will be modified.
…
int q = (int *)malloc(sizeof(int));
 // this call or other future malloc/free
 // calls will crash because the free
 // list is corrupted.
```

Double Free

- Double free is caused by freeing an object that is already free.
- This can cause the object to be added to the free list twice corrupting the free list.
- After a double free, future calls to malloc/free may crash.

Double Free

Example:

```
int * p = (int *)malloc(sizeof(int));
…
free(p); // free adds object to free list
…
free(p); // freeing the object again
         // overwrites the next/prev ptr
         // corrupting the free list
         // future calls to free/malloc
         // will crash
```

Wild Frees

- Wild frees happen when a program attempts to free a pointer in memory that was not returned by malloc.
- Since the memory was not returned by malloc, it does not have a header.
- When attempting to free this non-heap object, the free may crash.
- Also if it succeeds, the free list will be corrupted so future malloc/free calls may crash.

Wild Frees

- Also memory allocated with *malloc()* should only be deallocated with *free()* and memory allocated with new should only be deallocated with delete.
- Wild frees are also called “free of non-heap objects”.

Wild Frees

Example:

```
int q;
int * p = &q;

free(p);
 // p points to an object without
 // header. Free will crash or
 // it will corrupt the free list.
```

Wild Frees

Example:

```
char * p = (char *) malloc(1000);
p=p+10;

free(p);
 // p points to an object without
 // header. Free will crash or
 // it will corrupt the free list.
```

Memory Smashing

- Memory Smashing happens when less memory is allocated than the memory that will be used.
- This causes overwriting the header of the object that immediately follows, corrupting the free list.
- Subsequent calls to malloc/free may crash
- Sometimes the smashing happens in the unused portion of the object causing no damage.

Memory Smashing

Example:

```
 char * s = malloc(8);
strcpy(s, “hello world”);

  // We are allocating less memory for
  // the string than the memory being
  // used. Strcpy will overwrite the
  // header and maybe next/prev of the
  // object that comes after s causing
  // future calls to malloc/free to crash.
  // Special care should be taken to also
  // allocate space for the null character
  // at the end of strings.
```

Debugging Memory Allocation Errors

- Memory allocation errors are difficult to debug since the effect may happen farther away from the cause.
- Memory leaks is the least important of the problems since the effect take longer to show up.
- As a first step to debug premature free, double frees, wild frees, you may comment out free calls and see if the problem goes away.
- If the problem goes away, you may uncomment the free calls one by one until the bug shows up again and you find the offending free.

Debugging Memory Allocation Errors

There are tools that help you detect memory allocation errors.

- IBM Rational Purify
- Bounds Checker
- Insure++

Explicit Memory Allocation Advantages and Disadvantages

- Advantages:
  - It is fast and
  - It is memory efficient.
- Disadvantages
  - Error prone
  - Requires more expertise
  - Applications take longer to develop and debug.
  - More insecure
- This is why managers prefer now to develop in Java/C#.
- Still there is room for C/C++ for apps that need to be low-level and where speed is critical.

Some Improvements in Your Allocator

- You can add some checks to make your allocator more robust against memory errors.
- You can check if a block has been returned by malloc by storing a magic number in the header and checking it during free to prevent wild frees.
- Instead of only using one bit in the flags, you could use the entire 4 bytes to store a magic number to indicate if the object is free or not.
  - Free – 0xF7EEF7EE
  - Allocated – 0xAl0CA7ED
- When freeing an object check if in the header the flag contains the right magic number.

Heap Check

- You can implement also a heapCheck function that checks for the sanity of the heap and prints the free and allocated blocks.
- You can call it during debugging before and after suspicious free/malloc calls.
- Starting from the header at the beginning of the heap, and follow the headers. checkHeap iterates over all free and allocated blocks.
- To get around gaps in the heap, you can store the correct size of the gaps in the fence posts around the gaps.

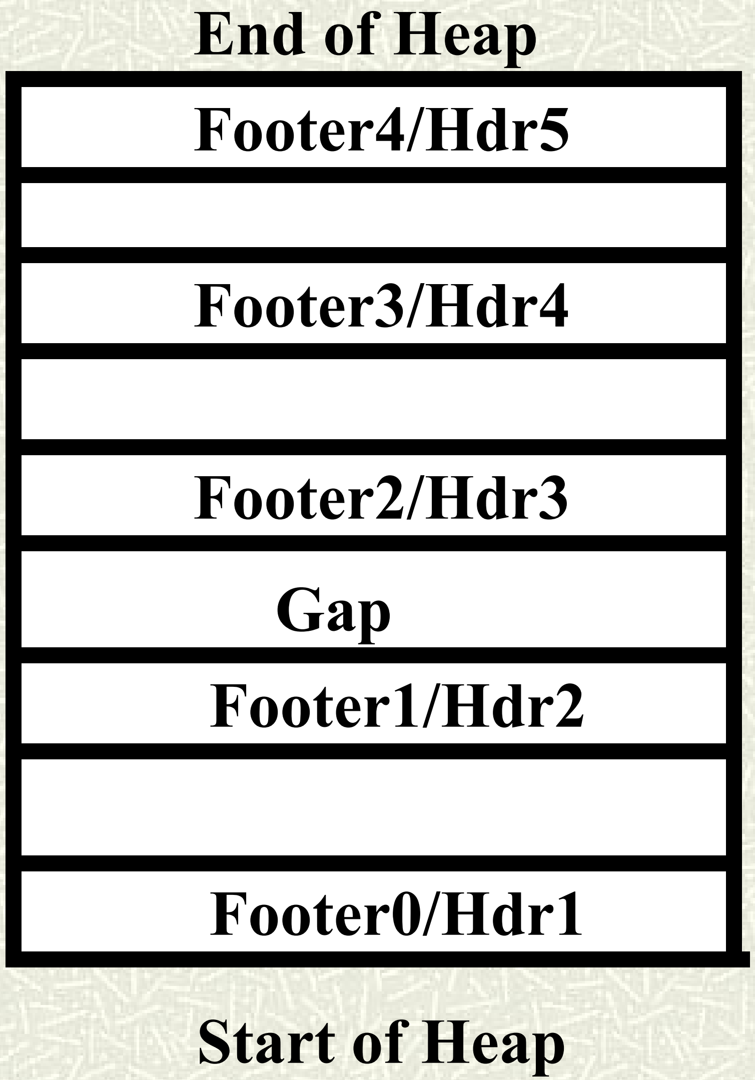
Heap Check

- There is enough information in the headers to iterat over the end of the heap.
- Verify that the sizes and flags in header and footer match.
- Also check that there are not two consecutive free blocks.

Heap Check

- Print also the blocks. Add an extra flag value to differentiate Gaps in the heap from allocated and free blocks.
- Also you may write a function printFreeBlocks() tha iterates over all lists and prints the free blocks.

Heap Check


End of Heap
Footer4/Hdr5
Footer3/Hdr4
Footer2/Hdr3
Gap
Footer1/Hdr2
Footer0/Hdr1
Start of Heap

Malloc Implementation Notes

```
// Header of an object. Used both when the object is allocated and freed
struct ObjectHeader {
    size_t _objectSize;         // Real size of the object.
    int _allocated;             // 1 = yes, 0 = no 2 = sentinel
    struct ObjectHeader * _next;// Points to the next object in the freelist (if free).
    struct ObjectHeader * _prev;// Points to the previous object.
};
struct ObjectFooter {
    size_t _objectSize;
    int _allocated;
};
char * chunk -= ….;
FreeObjectHeader  *fhdr = (FreeObjectHeader  *)chunk;
// Get fhdr->_objectSize, fhdr->prev, fhdr->next, etc
```

Adding Fence Posts

```
void initialize()
{…
 void * _mem = getMemoryFromOS( ArenaSize + (2*sizeof(struct ObjectHeader)) + (2*sizeof(struct ObjectFooter)) );
  // In verbose mode register also printing statistics at exit
  atexit( atExitHandlerInC );
  //establish fence posts
  struct ObjectFooter * fencepost1 = (struct ObjectFooter *)_mem;
  fencepost1->_allocated = 1;
  fencepost1->_objectSize = 123456789;
  char * temp =  (char *)_mem + (2*sizeof(struct ObjectFooter)) + sizeof(struct ObjectHeader) + ArenaSize;
  struct ObjectHeader * fencepost2 = (struct ObjectHeader *)temp;
  fencepost2->_allocated = 1;
  fencepost2->_objectSize = 123456789;
  fencepost2->_next = NULL;
  fencepost2->_prev = NULL;

// See lab1 handout
```

Using a Debugger

What is GDB

- GDB is a debugger that helps you debug your program.
- The time you spend now learning gdb will save you days of debugging time.
- A debugger will make a good programmer a better programmer.

Compiling a program for gdb

- You need to compile with the “-g” option to be able to debug a program with gdb.
- The “-g” option adds debugging information to your program

```
gcc –g –o hello hello.c
```

Running a Program with gdb

- To run a program with gdb type

```
gdb progname
(gdb)
```

- Then set a breakpoint in the main function.

```
(gdb) break main
```

- A breakpoint is a marker in your program that will make the program stop and return control back to gdb.
- Now run your program.

```
(gdb) run
```

- If your program has arguments, you can pass them after run.

Stepping Through your Program

- Your program will start running and when it reaches “main()” it will stop.

```
gdb>
```

- Now you have the following commands to run your program step by step:

```
(gdb) step
```

It will run the next line of code and stop. If it is a function call, it will enter into it

```
(gdb) next
```

It will run the next line of code and stop. If it is a function call, it will not enter the function and it will go through it.

Example:

```
(gdb) step
(gdb) next
```

Setting breakpoints

- You can set breakpoints in a program in multiple ways:

```
(gdb) break function
```

Set a breakpoint in a function E.g.

```
(gdb) break main
```

```
(gdb) break line
```

Set a break point at a line in the current file. E.g.

```
(gdb) break 66
```

It will set a break point in line 66 of the current file.

```
(gdb) break file:line
```

It will set a break point at a line in a specific file. E.g.

```
(gdb) break hello.c:78
```

Regaining the Control

- When you type

```
(gdb) run
```

  the program will start running and it will stop at a break point.

- If the program is running without stopping, you can regain control again typing ctrl-c.

Where is your Program

- The command

```
(gdb)where
```

Will print the current function being executed and the chain of functions that are calling that fuction.

This is also called the backtrace.

Example:

```
(gdb) where
#0  main () at test_mystring.c:22
(gdb)
```

Printing the Value of a Variable

- The command

```
(gdb) print var
```

Prints the value of a variable.

E.g.

```
(gdb) print i
$1 = 5
(gdb) print s1
$1 = 0x10740 "Hello"
(gdb) print stack[2]
$1 = 56
(gdb) print stack
$2 = {0, 0, 56, 0, 0, 0, 0, 0, 0, 0}
(gdb)
```

Exiting gdb

- The command “quit” exits gdb.

```
 (gdb) quit
The program is running.  Exit
  anyway? (y or n) y
```

Debugging a Crashed Program

- This is also called "postmortem debugging"
- It has nothing to do with CSI ☺
- When a program crashes, it writes a **core file**.

```
bash-4.1$ ./hello
Segmentation Fault (core dumped)
bash-4.1$
```

- The core is a file that contains a snapshot of the program at the time of the crash. That includes what function the program was running.

Debugging a Crashed Program

- Sometimes the sysadmins disable the generation of core files to reduce the disk space waste. This happens in the CS machines.
- To find out if your system is able to generate cores type:

```
grr@data ~/cs252 $ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
```

- If you see that the core file size is 0. Enable core file generation by typing:

```
grr@data ~/cs252 $ ulimit -c 1000000
```

Debugging a Crashed Program

- To run gdb in a crashed program type

  ```
  gdb program core
  ```

  E.g.

  ```
  bash-4.1$ gdb hello core
  GNU gdb 6.6
  Program terminated with signal 11, Segmentation fault.
  #0  0x000106cc in main () at hello.c:11
  11          *s2 = 9;
  (gdb)
  ```

  - Now you can type ***where*** to find out where the program crashed and the value of the variables at the time of the crash.

    ```
    (gdb) where
    #0  0x000106cc in main () at hello.c:11
    (gdb) print s2
    $1 = 0x0
    (gdb)
    ```

  - This tells you why your program crashed. Isn't that great?

Now Try gdb in Your Own Program

- Make sure that your program is compiled with the –g option.
- Remember:
  - One hour you spend learning gdb will save you days of debugging.
  - Faster development, less stress, better results

The UNIX Operating System

What is an Operating System

- An Operating System (OS) is a program that sits in between the hardware and the user programs.
- It provides:
  - Multitasking - Multiple processes running in the same computer
  - Multiuser - Multiple users using the same computer
  - File system – Storage
  - Networking – Access to the network and internet

What is an Operating System

- Window System – Graphical use interface
- Standard Programs – Programs such as a web browser, task manager, editors, compilers etc.
- Common Libraries – Libraries common to all programs running in the computer such as math library, string library, window library, c library etc.
- It has to do all of the above in a secure and reliable manner.

A Tour of UNIX

- We will start by describing the UNIX operating system (OS).
- Understanding one instance of an Operating System will help us understand other OSs such as Windows, Mac OS, Linux etc.
- UNIX is an operating system created in 1969 by Ken Thompson, Dennis Ritchie, Brian Kernighan, and others at AT&T Bell Labs.
- UNIX was a successor of another OS called MULTICS that was more innovative but it had many problems.
- UNIX was smaller, faster, and more reliable than MULTICS.

A Tour of UNIX

- UNIX was initially created to support typesetting (edition of documents).
- By having the programmers being the users themselves of the OS (it your own food), UNIX became the robust, practical system that we know today.
- UNIX was written in “C” (95%) and assembly language (5%).
- This allowed UNIX to be ported to other machines besides Digital Equipment (DEC)’s PDP11.

BSD UNIX

- UNIX was a success in the universities.
- Universities wanted to modify the UNIX sources for experimentation do Berkeley created its own version of UNIX called BSD-UNIX.
- POSIX is an organization that created the POSIX UNIX standard to unify the different flavors of UNIX.
- Sockets, FTP, Mail etc came from BSD UNIX.

The UNIX File System

UNIX File System

- UNIX has a hierarchical File System
- Important directories

  / - Root Directory

  /etc OS Configuration files

    /etc/passwd – User information

    /etc/groups – Group information

    /etc/inetd.conf – Configuration of Internet Services (deamons)

    /etc/rc.*/ - OS initialization scripts for diffeerent services.

- Deamons – Programs running in the background implementing a service. (Servers).

UNIX File System

- /dev – List of devices attached to the computer
- /usr – Libraries and tools
  - /usr/bin – Application programs such as grep, ls et
  - /usr/lib – Libraries used by the application programs
  - /usr/include – Include files (.h) for the libraries
- /home – Home directories

Users

- UNIX was designed as a multiuser system.
- The database of users is in /etc/passwd

```
lore 2 % cat /etc/passwd | grep grr
    grr:x:759:759:Gustavo Rodriguez
 Rivera,,,:/homes/grr:/bin/tcsh
```

- Each line has the format: login:userid:groupid:Name,,,:homedir:shell
- Every user has a different “USER ID” that is a number that identifies the user uniquely in the system.
- The encrypted password used to be stored also here. Now it is stored in /etc/shadow

Users

- Commands for users
  - adduser – Adds a new user
  - passwd – Change password.
- There exist a special user called “root” with special privileges.
- Only root can modify files anywhere in the system.
- To login as root (superuser) use the command “su”.
- Only root can add users or reset passwords.

Groups

- A “group” represents a group of users.
- A user can belong to several groups.
- The file /etc/group describes the different groups in the system.

Yellow Pages

- In some systems the password and group files is stored in a server called “Yellow Pages” that makes the management easier.
- If your UNIX system uses yellow pages the group and database are in a server. Use “ypcat”

```
ypcat group | grep cs240
lore 15 % ypcat group | grep cs240
cs240:*:15196:crisn,grr,rego,yau
```

- Also the passwd file can be in Yellow Pages:

```
lore 16 % ypcat passwd | grep grr
grr:##grr:759:759:Gustavo Rodriguez-Rivera,,,:/homes/grr:/bin/tcsh
```

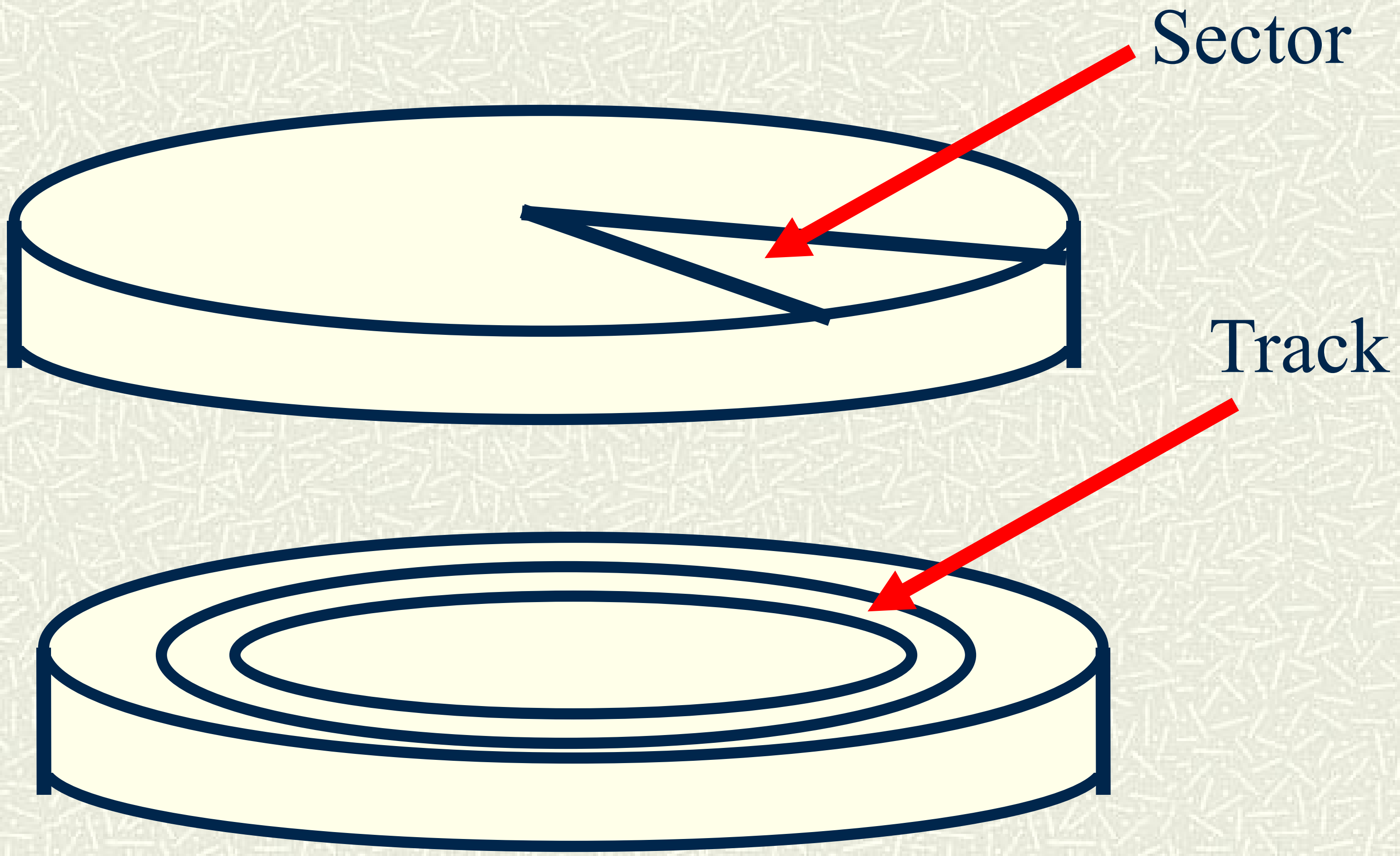
File Systems

- The storage can be classified from fastest to slowest in the following
  - Registers
  - Cache
  - RAM
  - Flash Memory
  - Disk
  - CD/DVD
  - Tape
  - Network storage

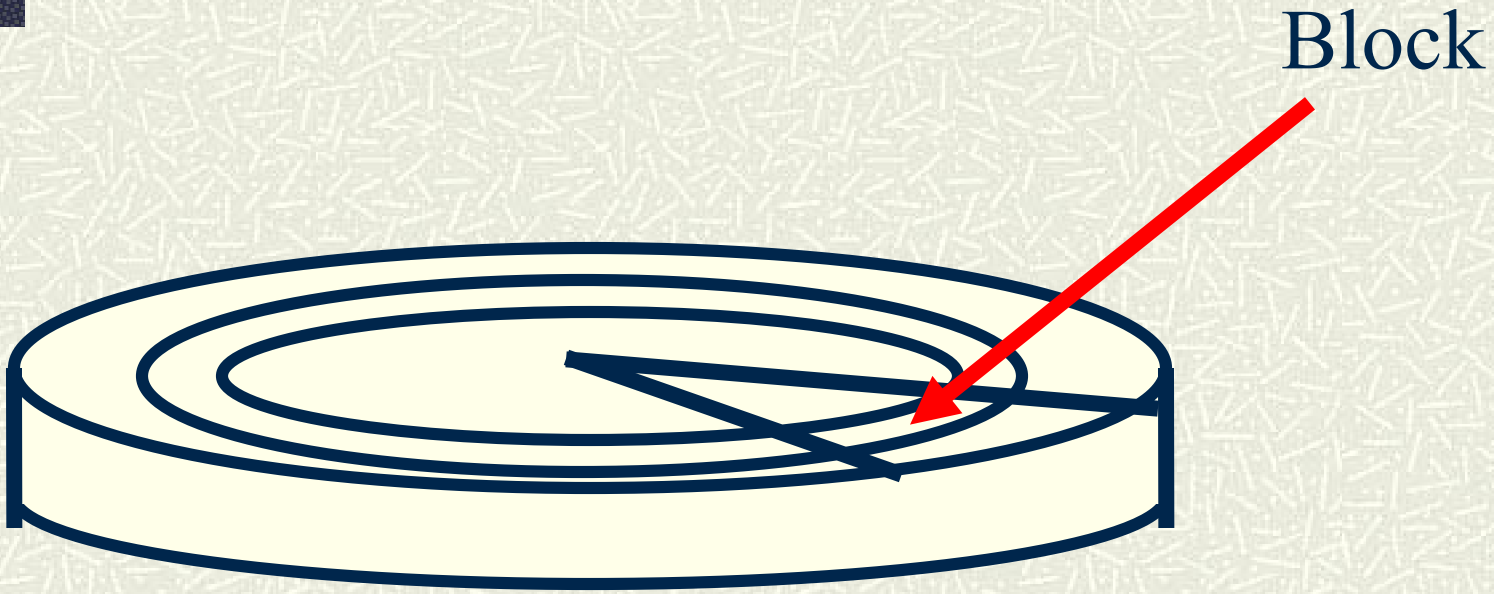
Disk File Systems

- The disk is a an electromagnetic and mechanical device that is used to store information permanently.
- The disk is divided into sectors, tracks and blocks

Disk File Systems



Disk File Systems

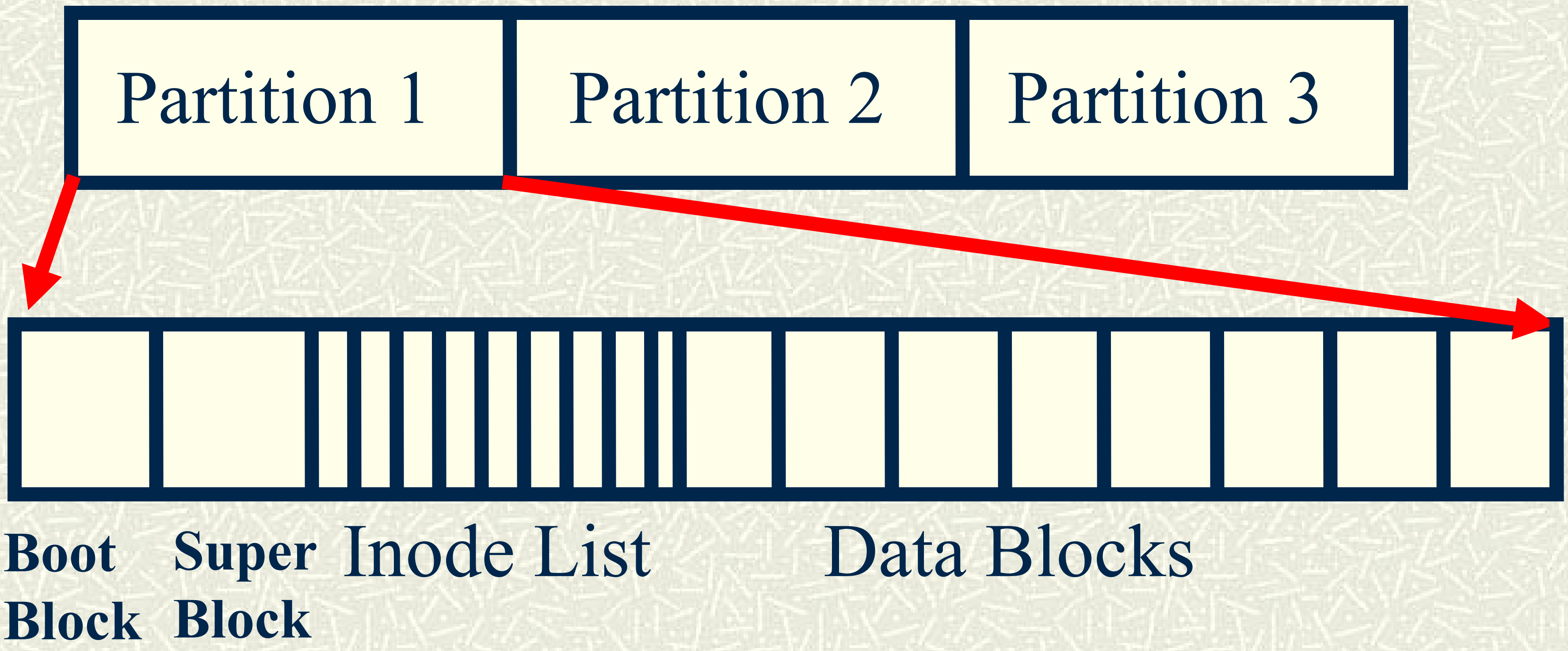


A Block is the intersection between a sector and a track

Disk File Systems

- Disks when formatted are divided into sectors, tracks and blocks.
- Disks are logically divided into partitions.
- A partition is a group of blocks.
- Each partition is a different file system.

Disk File System



Disk File System

- Each partition is divided into:
  - **Boot Block** – Has a piece of code that jumps to the OS for loading.
  - **Superblock** – Contain information about the number of data blocks in the partition, number of inodes, bitmap for used/free inodes, and bitmap for used/free blocks, the inode for the root directory and other partition information.
  - **Inode-list** – It is a list of I-nodes. An inode has information about a file and what blocks make the file. There is one inode for each file in the disk.
  - **Data Blocks** – Store the file data.

I-node information

- An i-node represents a file in disk. Each i-node contains:
  1. Flag/Mode
     1. Read, Write, Execute (for Owner/Group/All) RWX RWX RWX

     Also tells if file is directory, device, symbolic link
  2. Owners
     1. Userid, Groupid
  3. Time Stamps
     1. Creation time, Access Time, Modification Time.
  4. Size
     1. Size of file in bytes
  5. Ref. Count –
     1. Reference count with the number of times the i-node appears in a directory (hard links).
     2. Increases every time file is added to a directory. The file the i-node represents will be removed when the reference count reaches 0.

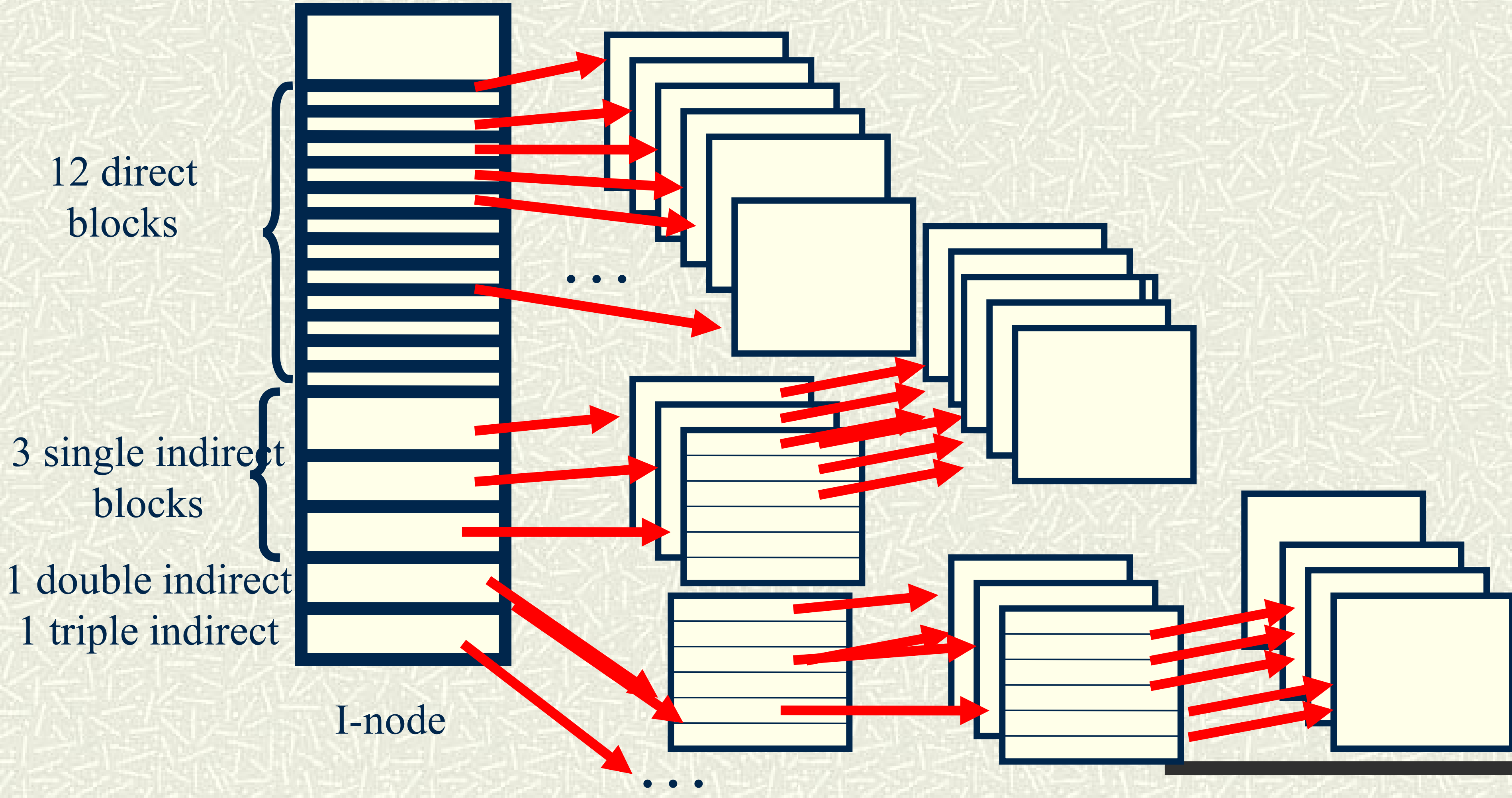
I-node information

- The I-node also contains a block index with the blocks that form the file.
- To save space, the block index uses indices of different levels.
- This benefits small files since they form the largest percentage of files.
- Small files only uses the direct and single-indirect blocks.
- This saves in space spent in block indices.

I-node information

- Direct block –
  - Points directly to the block. There are 12 of them in the structure
- Single indirect –
  - Points to a block table that has 256 entry's. There are 3 of them.
- Double indirect –
  - Points to a page table of 256 entries which then points to another page table of 256
- Triple Indirect
  - Points to a page table of 256 entries which then points to another page table of 256 that points to another page of 256 bytes.

I-node Block Index



I-node information

- Assume 1KB block and 256 block numbers in each index block.
- Direct block = 12 * 1Kb = 12Kb
- Single indirect = 3 * 256 * 1Kb = 768 Kb
- Double indirect = 1 * 256 * 256 * 1Kb = 64 Mb
- Triple indirect = 1 * 256 * 256 * 256 * 1Kb = 16 Gb

I-node information

- Most of the files in a system are small.
- This also saves disk access time since small files need only direct blocks.
  - 1 disk access for the I-Node
  - 1 disk access for the datablock.
- An alternative to the multi-level block index is a ***linked list***. Every block will contain a pointer to the next block and so on.
- Linked lists are slow for random access.

Directory Representation and Hard Links

- A directory is a file that contains a list of pairs (file name, I-node number)
- Each pair is also called a hard-link
- An I-node may appear in multiple directories.
- The reference count in the I-node keeps track of the number of directories where the I-node appears.
- When the reference-count reaches 0, the file is removed.

Hard Links

- In some OSs, the reference count is incremented when the file is open.
- This prevents the file from being removed while it is in use.
- Hard Links cannot cross partitions, that is, a directory cannot list an I-node of a different partition.
- Example. Creating a hard link to a target-file in the current directory

  ln target-file name-link

Soft-Links

- Directories may also contain Soft-Links.
- A soft -link is a pair of the form
  (file name, i-node number-with-file-storing-path)

  Where path may be an absolute or relative path in this or another partition.
- Soft-links can point to files in different partitions.
- A soft -link does not keep track of the target file.
- If the target file is removed, the symbolic link becomes invalid (dangling symbolic link).
- Example:

  ln –s target-file name-link

File Ownership

- The Group Id and owner’s User ID are stored as part of the file information
- Also the creation, modification, and access time are stored in the file in addition to the file size.
- The time stamps are stored in seconds after the Epoch (0:00, January 1st, 1970).

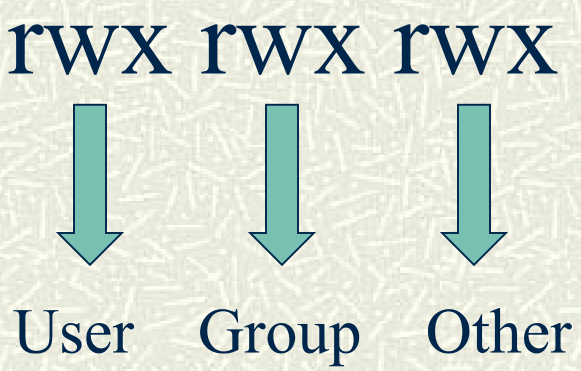
File Permissions

- The permissions of a file in UNIX are stored in the inode in the flag bits.
- Use “ls –l” to see the permissions.

```
-rw-rw-r--    1 grr            150 Aug 29  1995 calendar
-rw-------    1 grr            975 Mar 25  1999 cew.el
-rwxrwxr-x    1 grr           5924 Jul  9 10:48 chars
-rw-rw-r--    1 grr            124 Jul  9 10:47 chars.c
drwxr-sr-x   10 grr            512 Oct 14  1998 contools
drwxr-sr-x    9 grr            512 Oct  8  1998 contools-new
```

Permission Bits

The permissions are grouped into three groups: User, Group, and Others.



Permission Bits

- To change the persmissions of a file use the command chmod.

  chmod <u|g|o><+|-><r|w|x>

  Where

  <u|g|o> is the owner, group or others.

  <+|-> Is to add or remove permissions

  <r|w|x> Are read, write, execute permissions.

- Example

Permission Bits Example

- Make file “hello.txt” readable and writable by user and group but only readable by others

```
chmod u+rw hello.txt
chmod g+rw hello.txt
chmod o+r hello.txt
chmod o-w hello.txt
```

  - Scripts and Executable files should have the executable bit set to be able to execute them.

```
chmod ugo+x myscript.sh
```

Permission Bits

- Also you can change the permission bits all at once using the bit representation in octal

```
USER GROUP OTHERS
RWX    RWX    RWX
110    110    100     - Binary
 6      6      4      - Octal digits
```

```
chmod 664 hello.c
```

Directory Bit

- The Directory Bit in the file flags indicates that the file is a directory
- When an file is a directory the “x” flag determines if the file can be listed or not.
- If a file has its directory with “+x” but not readable “-r” then the file will be accessible but it will be invisible since the directory cannot be listed.

Process’ Properties

- A process has the following properties:
  - PID: Index in process table
  - Command and Arguments
  - Environment Variables
  - Current Dir
  - Owner (User ID)
  - Stdin/Stdout/Stderr

Process ID

- Uniquely identifies the processes among all live processes.
- The initial process (init process) has ID of 0.
- The OS assigns the numbers in ascending order.
- The numbers wrap around when they reach the maximum and then are reused as long as there is no live process with the same processID.
- You can programmatically get the process id with
  - int getpid();

Command and Arguments

- Every process also has a command that is executing (the program file or script) and 0 or more arguments.
- The arguments are passed to main.

```
int main(int argc, char **argv);
```

- Argc contains the number of arguments including the command name.
- Argv[0] contains the name of the command

Printing the Arguments

```
printargs.c:
int main(int argc, char **argv) {
  int i;
  for (i=0; i<argc; i++) {
    printf("argv[%d]=\"%s\"\n", i, argv[i]);
  }
}

gcc -o printargs printargs.c
./printargs hello world
argv[0]="./printargs"
argv[1]="hello"
argv[2]="world"
```

Environment Variables

- It is an array of strings of the form A=B that is inherited from the parent process.
- Some important variables are:
  - PATH=/bin:/usr/bin:. Stores the list of directories that contain commands to execute.
  - USER=<login> Contains the name of the user
  - HOME=/homes/grr Contains the home directory.
- You can add Environment variables settings in .login or .bashrc and they will be set when starting a shell session.

Environment Variables

- To set a variable from a shell use

  export A=B

  - Modify the environment globally. All processes called will get this change

  A=B

  – Modify environment locally. Only current shell process will get this change.

- Example: Add a new directory to PATH

  export PATH=$PATH:/newdir

Printing Environment

- To print environment from a shell type "env".

```
lore 24 % env
USER=grr
LOGNAME=grr
HOME=/homes/grr
PATH=/opt/csw/bin:/opt/csw/gcc3/bin:/p/egcs-
   1.1b/bin:/u/u238/grr/Orbix/bin:/usr/local/gnu:/p/srg/bin
   :/usr/ccs/bin:/usr/local/bin:/usr/ucb:/bin:/usr/bin:/usr
   /hosts:/usr/local/X11:/usr/local/gnu:.
MAIL=/var/mail/grr
SHELL=/bin/tcsh
TZ=US/East-Indiana
…
```

Printing Environment from a Program

- r through the “char ** environ” variable.
- environ points to an array of strings of the form A=B and ends with a NULL entry.

```
char **environ;
int main(int argc, char **argv) {
  int i=0;
  while (environ[i]!=NULL) {
    printf("%s\n",environ[i]);
    i++;
  }
}
```

Current Directory

- Every process also has a current directory.
- The open file operations such as open() and fopen() will use the current directory to resolve relative paths.
- If the path does not start with “/” then a path is relative to the current directory.

  /etc/hello.c – Absolute path

  hello.c – Relative path.

To change the directory use “cd dir” in a shell or chdir(dir) inside a program

Process’ User ID

- A process always runs in behalf of a user represented by the User ID.
- The UID is inherited from the parent process.
- Only root can change the UID of a process at runtime using the “setuid(uid);” call.
- This happens at login time. The login program runs as root but then after identifying the user, the OS switches the UID to that user and runs the shell.
- It also can be done with the comand “su user” that prompts for that user’s password.
- Also “sudo user command” runs the command as that user.

Stdin/Stdout/Stderr

- Also a process inherits from the parent a stdin/stdout and stderr.
- They are usually the keyboard and the terminal but they can be redirected.
- Example:

```
command < in.txt > out.txt 2> err.txt
```

- From a program you can redirect stdin,stdout,stderr using dup(), and dup2(). We will cover that more in depth later.

Redirection of stdin/stdout/stderr

```
command >> out
```

Append output of the command into out.

```
command > out.txt 2> err.txt
```

Redirect stdout and stderr.

```
command > out.txt 2>&1
```

Redirect both stderr and stdout to file out.txt

PIPES

- In UNIX you can connect the output of a command to the input of another using PIPES.

Example:

```
ls –al | sort
```

Lists the files in sorted order.

Common UNIX Commands

Common UNIX Commands

- There are many UNIX commands that can be very useful
- They output of one can be connected to the input of the other using PIPES
- They are part of the UNIX distribution.
- Open source implementations exist and they are available in Linux.

ls – list files and directories

- ls <options> file list
- Examples

  ls –al

  - Lists all files including hidden files (files that start with ".") and timestamps.

  ls –R dir

  - Lists recursively all directories and subdirectories in dir.

mkdir – Make a directory

- mkdir <options> dir1 …
- Examples

  mkdir dir1

    Create directory dir1

  mkdir –p dir1/dir2/dir3

    Make parent directory an subdirectories if it they do not exist.

cp – Copy files

- cp <options> file1 file2 … destdir

  Copies one or more files to a destination.
- Examples:

  cp a.txt dir1

  Copies file a.txt into dir1

  cp a.txt b.txt

  Create a copy of a.txt called b.txt

  cp –R dir1 dir2

  Copy recursively directories and subdirectories of dir1 into dir2.

mv – Move a file

- mv file1 destdir

  Moves file1 to destdir
- Examples:

  mv a.txt dir1

    Move file a.txt into directory dir1

  mv a.txt b.txt

    Rename file a.txt into b.txt

rm – Remove a file

- rm <options> file1 file2 …

  Removes a list of files

- Examples:

  rm a.txt b.txt

    Remove files a.txt and b.txt

  rm –f a.txt

    Remove a.txt. Do not print error message if fails.

  rm –R dir1

    Remove dir1 and all its contents.

grep – Find lines

- grep <options> pattern file1 file2 …
  - Print lines that contain “pattern”
- Examples:

  grep hello a.txt

  Print the lines in a.txt that contain hello

  grep hello * */* */*/*

  Print the lines that contain hello in any file in the directory and subdirectories.

man – Print manual pages

- man <options> command
  - Print the manual page related to command.
- Examples:

  man cp

  Print the manual pages related to copy

  man –k pthread

  Print all manual pages that contain string “pthread”.

  man –s 3 exec

  Print manual page of exec from section 3
- Man Pages are divided into sections:

  Section 1 – UNIX commands (E.g. cp, mv etc)

  Section 2 – System Calls (E.g. fork)

  Section 3 – C Standard Library

  Example “man –s 1 printf” and “man –s 3c printf” give different man page. One printf from the shell, and the other from the lib C library.

whereis – Where a file is located

- Where file
  - Prints the path of where a file is located.
  - It only works if the OS created a database with the files in the file system.
  - Example

```
bash-4.1$ whereis apachectl
/p/apache/apachectl
  (PATH=/p/apache)
/p/apache-php/bin/apachectl
  (PATH=/p/apache-php/bin)
/p/apache/man/man8/apachectl.8
  (MANPATH=/p/apache/man)
/p/apache-php/man/man8/apachectl.8
  (MANPATH=/p/apache-php/man)
```

which – Path of a command

- which command
  - Prints the path of the command that will be executed by the shell if “command” is typed.

Example:

```
bash-4.1$ which ps
/usr/ucb/ps
bash-4.1$ whereis ps
/usr/bin/ps                                 (PATH=/usr/bin)
/usr/ucb/ps                                 (PATH=/usr/ucb)
bash-4.1$ export PATH=/usr/bin:$PATH
bash-4.1$ which ps
/usr/bin/ps
```

head – Lists the first few lines

- head file

  List the first 10 lines of a file

Example:

```
head myprog.c
```

Lists the first 10 lines of myprog.c

```
head -5 myprog.c
```

List the first 5 lines of myprog.

tail – Lists the last few lines

- tail file
  - List the last 10 lines of a file
- Example:

```
tail myprog.c
```

list the last 10 lines of myprog.c

```
tail -3 myprog.c
```

list the last 3 lines of myprog.c

```
tail –f a.log
```

It will periodically print the lines of a.log as they are added.

awk – pattern scanning and processing language

- An awk is a txt processing program.
- The program file has a sequence of rules of the form:

  pattern {action}

  …

  default {action}

  Where pattern is a regular expression. And action is a sequence of statements that run when the pattern is matched with the text.

Awk examples

- You can also use awk for simple text manipulation:

Example:

```
awk '{print $1;}' m.txt
```

print the first word of each line in file m.txt

Google awk for more information

sed – A simple line editor

- Used for simple test processing
- Example:

```
sed s/current/new/g hello.sh > hello2.sh
```

  Replaces all the instances of current to new and redirects the output to hello2.sh

find – Execute a command for a group of files

- find – Recursively descends the directory hierarchy for each path seeking files that match a Boolean expression.
- Examples:

```
find . -name "*.conf" -print
```

Find recursively all files with .conf extension.

```
find . -name "*.conf" -exec chmod o+r
'{}' \;
```

Find recursively and make readable by others all files with .conf extension

Shell Scripting

Shell Programs

- Shells are programs used to interact with a computer using a command line.
- They used to be the only way that users could interact with the computer.
- Now there are Graphical Shells like Gnome, KDE, Windows Explorer, Macintosh Finder, etc that offer similar functionality.
- GUI Shells can do many of the tasks of command line shells but not all of them.
- Shells are used to make many administration procedures automatic: backup, program installation, Web services.

Shell Programs

- When a command runs, the OS checks first if the file is an executable file with a known format (ELF, a.out).
- If it is not, then it checks the first line for a “#!interpreter-program”, if it is there, then it runs the interpreter program and passes the file as input.
- In any case, a command needs to have the execute permissions set to be able to run.

Shell Programs

- /bin/sh – Standard UNIX Shell
- /bin/ksh – Korn shell (more powerful)
- /bin/bash – GNU shell
- /bin/tcsh – Some line editing.

- Bash is becoming widely available in the UNIX word since it is the standard Linux shell.

hello.sh - Example of a Shell Program

```
#!/bin/bash
#
# This shell script prints some data about the
  user
# and shows how to use environment variables
#
echo "Hello $USER"
echo "Welcome to CS252"
echo "Your home directory is \"$HOME\""
echo "And your current directory is \"$PWD\""
```

hello-loop.sh – Another example

```
#!/bin/bash
#
# This shell script prints hello to all the friends you
# pass as parameter
#

if [ $# -lt 1 ]
then
  echo
  echo "$0 needs at least one argument"
  echo "  Eg."
  echo "     $0 Mickey Donald Daisy”
  exit 1
fi

for friend in $*
do
  echo "Hello $friend"
done
```

mail-hello.sh - Send e-mail

```
#!/bin/sh
#
# This script builds a simple message and mails it to
# yourself
#
echo "Hello $USER!!" > tmp-message
echo >> tmp-message
echo "Today is" `date`  >> tmp-message
echo >> tmp-message
echo "Sincerely," >> tmp-message
echo "     Myself" >> tmp-message
/usr/bin/mailx -s "mail-hello" $USER < tmp-message
echo "Message sent."
```

count-files.sh - Shell to Count Files

```
#!/bin/bash
#
# Counts how many files are in the directories passed
# as parameter. If not directories are passed it uses
# the current directory.

# If no arguments use only current directory
if [ $# -lt 1 ]
then
  dirs=.
else
  dirs=$*
fi

#Initialize file counter to 0
count=0
```

Shell to count files (cont.)

```
# for all the directories passed as argument
for dir in $dirs
do
   echo $dir:
   for file in $dir/*
   do
       echo "$count: $file"
       count=`expr $count + 1`
   done
done

echo "$count files found"
```

Unix System Programming and The Shell Project

UNIX Organization

- UNIX has multiple components
  - Scheduler – Schedules processes
  - File System – Provides storage
  - Virtual Memory - Allows each process to have its own address space
  - Networking Subsystem
  - Windowing System
  - Shells and applications

Shell Project

- To interact with the OS you use a shell program or command interpreter
  - Csh – C Shell
  - Tcsh – Enhanced C Shell
  - Sh - Shell
  - Ksh – Korn Shell
  - Bash – GNU shell
- There are also other graphical shells like
  - Windows Desktop
  - Mac OS Finder
  - X Windows Managers

Shell Interpreter

- The shell project is divided into several subsystems:
- ***Parser***: reads a command line and creates a command table. One entry corresponds to a component in the pipeline.

  Example:

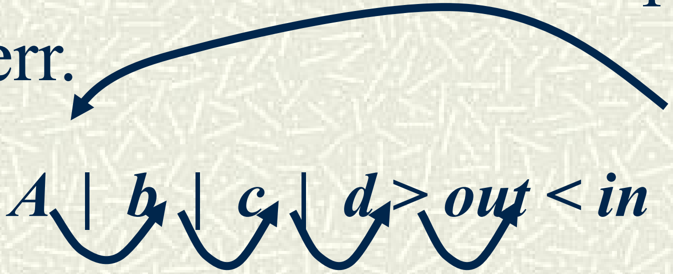
  Command: ls –al | grep me > file1

Command Table

ls	**-al**	
grep	**me**	
In:dflt	**Out:file1**	**Err:dflt**

Shell Interpreter

- Executor:
  - Creates new process for each entry in the command table.
  - It also creates pipes to communicate the output of one process to the input of the next one.
  - Also it redirects the stdinput, stdoutput, and stderr.



•All pipe entries share the same stderr

Shell Interpreter

- Other Subsystems
  - Environment Variables: Set, Print, Expand env vars
  - Wildcards: Arguments of the form a*a are expanded to all the files that match them.
  - Subshells: Arguments with ``(backticks) are executed and the output is sent as input to the shell.

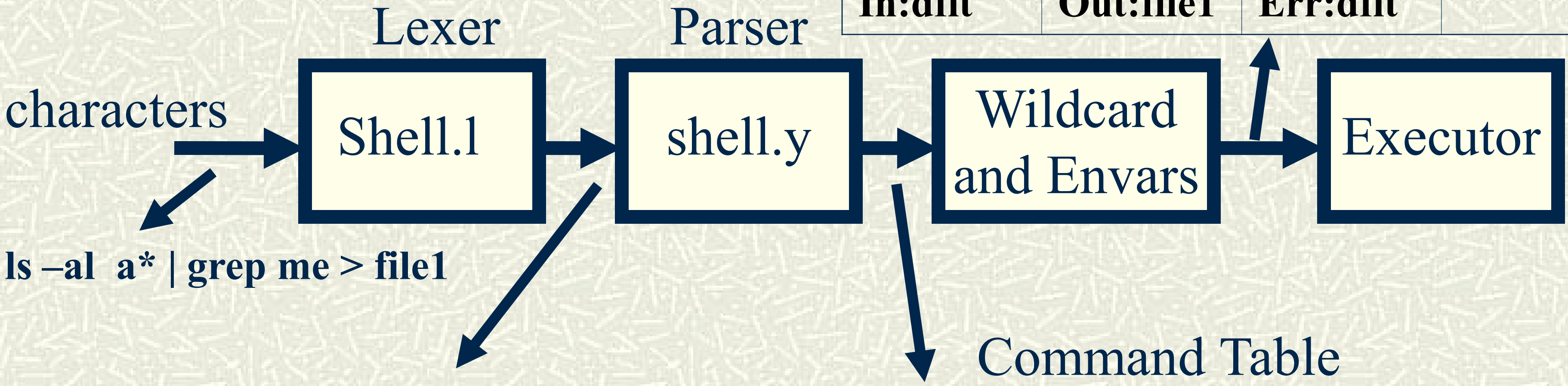
Shell Project

- Part 1: Shell Parser.
  - Read Command Line and print Command Table
- Part 2: Executer:
  - Create Processes and communicate them with pipes. Also do in/out/err redirection.
- Part 3: Other Susystems:
  - Wildcard, Envars, Subshells

Lex and Yacc

- A parser is divided into a ***lexical analyzer*** that separates the input into tokens and a ***parser*** that parses the tokens according to a grammar.
- The tokens are described in a file ***shell.l*** using regular expressions.
- The grammar is described in a file shell.y using syntax expressions.
- Shell.l is processed with a program called lex that generates a lexical analyzer.
- Shell.y is processed with a program called yacc that generates a parser program

Shell Project



ls –al a* | grep me > file1

<ls> <–al>
<a*> <PIPE>
<grep> <me>

<GREAT>

Final Command Table

ls	**-al**	**aab**	**aaa**
grep	**me**		
In:dflt	**Out:file1**	**Err:dflt**	

Command Table

ls	**-al**	**a***
grep	**me**	
In:dflt	**Out:file1**	**Err:dflt**

Shell Grammar

- You need to implement the following grammar in shell.l and shell.y

cmd [arg]* [| cmd [arg]*]* [[> filename] [< filename] [>& filename] [>> filename] [>>& filename]]* [&]

- Currently, the grammar implemented is very simple
- Examples of commands accepted by the new grammar.

```
ls –al
ls –al > out
ls –al | sort >& out
awk –f x.awk | sort –u < infile > outfile &
```

Lexical Analyzer

- Lexical analyzer separates input into tokens.
- Currently shell.l supports a reduced number of tokens
- ***Step 1***: You will need to add more tokens needed in the new grammar that are not currently in ***shell.l*** file

```
">>" { return GREATGREAT; }
"|" { return PIPE;}
"&" { return AMPERSAND}
```

Etc.

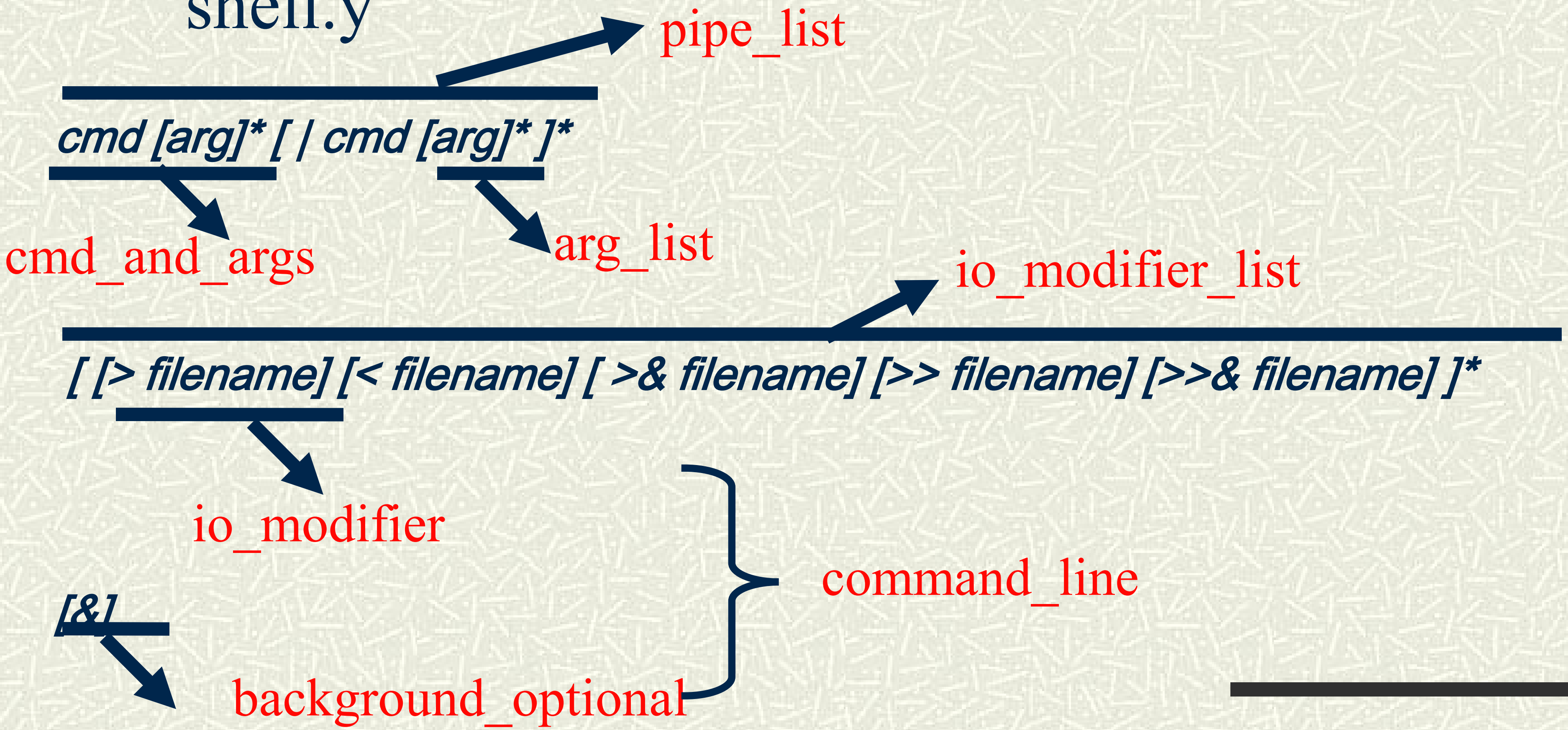
Shell Parser

- ***Step 2***. Add the token names to shell.y

%token NOTOKEN, GREAT, NEWLINE, WORD, GREATGREAT, PIPE, AMPERSAND etc

Shell Parser Rules

- Step 3. You need to add more rules to shell.y


pipe_list
cmd [arg]* [| cmd [arg]*]*
cmd_and_args
arg_list
io_modifier_list
[[> filename] [< filename] [>& filename] [>> filename] [>>& filename]]*
io_modifier
[&]
background_optional
command_line

Shell Parser Rules

```
goal: command_list;
arg_list:
    arg_list WORD
    | /*empty*/
    ;
cmd_and_args:
    WORD arg_list
    ;
```

Shell Parser Rules

```
pipe_list:
      pipe_list PIPE cmd_and_args
      | cmd_and_args
      ;
```

Shell Parser Rules

```
io_modifier:
     GREATGREAT Word
    | GREAT Word
    | GREATGREATAMPERSAND Word
    | GREATAMPERSAND Word
    | LESS Word
  ;
```

Shell Parser Rules

```
io_modifier_list:
    io_modifier_list io_modifier
    | /*empty*/
    ;
background_optional:
    AMPERSAND
    | /*empty*/
    ;
```

Shell Parser Rules

```
command_line:
      pipe_list io_modifier_list
        background_opt NEWLINE
     | NEWLINE /*accept empty cmd line*/
     | error NEWLINE{yyerrok;}
                 /*error recovery*/
command_list :
      command_line |
      command_list command_line
     ;/* command loop*/
```

Shell Parser Rules

- This grammar implements the command loop in the grammar itself.
- The ***error*** token is a special token used for error recovery. ***error*** will parse all tokens until a token that is known is found like <NEWLINE>. Yyerrok tells parser that the error was recovered.
- You need to add actions {…}in the grammar to fill up the command table. Example:

```
arg_list:
      arg_list WORD{currsimpleCmd->insertArg($2);}
      | /*empty*/
       ;
```

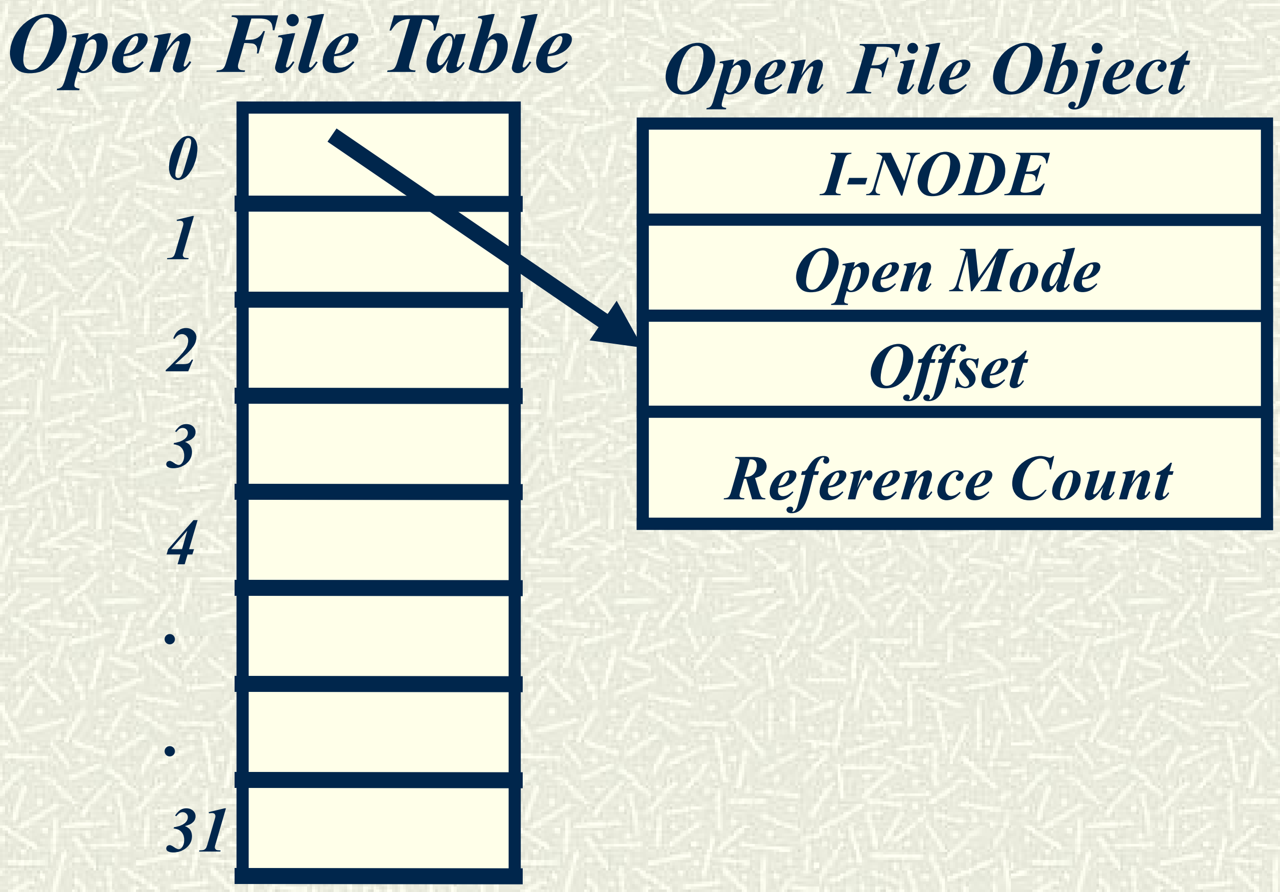
The Open File Table

- The process table also has a list with all the files that are opened
- Each open file descriptor entry contain a pointer to an open file object that contains all the information about the open file.
- Both the ***Open File Table*** and the ***Open File Objects*** are stored in the kernel.

The Open File Table

- The system calls like write/read refer to the open files with an integer value called ***file descriptor*** or ***fd*** that is an index into the table.
- The maximum number of files descriptor per process is 32 by default but but it can be changed with the command ***ulimit*** up to 1024.

The Open File Table



Open File Object

- An ***Open File Object*** contains the state of an open file.
  - I-Node –
    - It uniquely identifies a file in the computer. An I-nodes is made of two parts:
    - Major number – Determines the devices
    - Minor number –It determines what file it refers to inside the device.
  - Open Mode – How the file was opened:
    - Read Only
    - Read Write
    - Append

Open File Object

- Offset –
  - The next read or write operation will start at this offset in the file.
  - Each read/write operation increases the offset by the number of bytes read/written.
- Reference Count –
  - It is increased by the number of file descriptors that point to this Open File Object.
  - When the reference count reaches 0 the Open File Object is removed.
  - The reference count is initially 1 and it is increased after fork() or calls like dup and dup2.

Default Open Files

- When a process is created, there are three files opened by default:
  - 0 – Default Standard Input
  - 1 – Default Standard Output
  - 2 – Default Standard Error

  write(1, “Hello”, 5) Sends Hello to stdout

  write(2, “Hello”, 5) Sends Hello to stderr
- Stdin, stdout, and stderr are inherited from the parent process.

The ***open()*** system call

- int open(filename, mode, [permissions]),
  - It opens the file in ***filename*** using the permissions in ***mode***.
  - Mode:
    - O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_APPEND, O_TRUNC
  - O_CREAT If the file does not exist, the file is created.Use the permissions argument for initial permissions. Bits: rwx(user) rwx(group) rwx (others) Example: 0555 – Read and execute by user, group and others. (101B==5Octal)
  - O_APPEND. Append at the end of the file.
  - O_TRUNC. Truncate file to length 0.
  - See “man open”

The *close()* System call

- void close(int fd)
  - Decrements the count of the ***open file object*** pointed by fd
  - If the reference count of the ***open file object*** reaches 0, the open file object is reclaimed.

The *fork()* system call

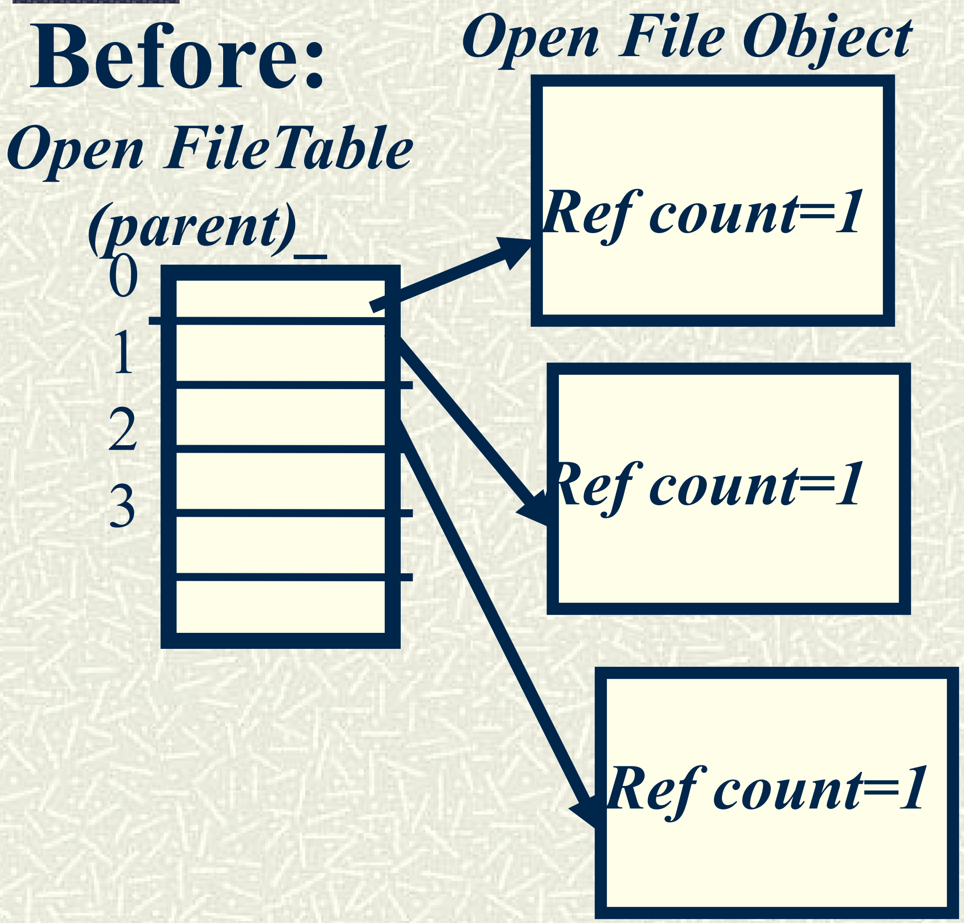
- ***int fork()***
  - It is the only way to create a new process in UNIX
  - The OS creates a new child process that is a copy of the parent.
  - ret = fork() returns:
    - ret == 0 in the child process
    - ret == pid > 0 in the parent process.
    - ret < 0 error
  - The memory in the child process is a copy of the parent process's memory.
  - We will see later that this is optimized by using VM copy-on-write.

The *fork()* system call

- The Open File table of the parent is also copied in the child.
- The Open File Objects of the parent are shared with the child.
- Only the reference counters of the Open File Objects are increased.

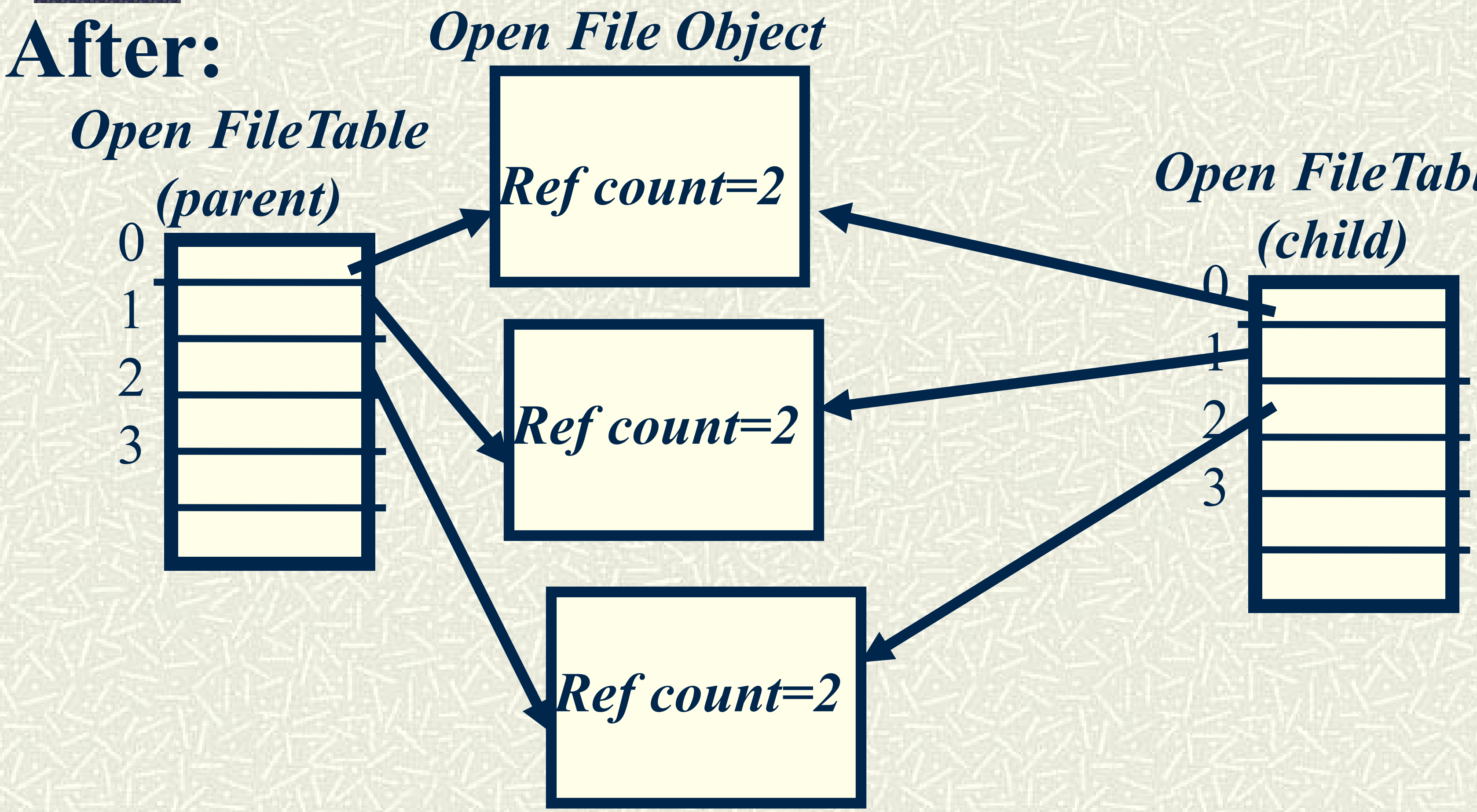
The *fork()* system call

Before:



The *fork()* system call

After:



The *fork()* system call

- Implication of parent and child sharing file objects:
  - By sharing the same open file objects, parent and child or multiple children can communicate with each other.
  - We will use this property to be able to make the commands in a pipe line communicate with each other.

The execvp() system call

- int execvp(progname, argv[])
  - Loads a program in the current process.
  - The old program is overwritten.
  - ***progname*** is the name of the executable to load.
  - ***argv*** is the array with the arguments. Argv[0] is the progname itself.
  - The entry after the last argument should be a NULL so ***execvp()*** can determine where the argument list ends.
  - If successful, execvp() will not return.

The execvp() system call

- Example: Run “ls –al” from a program.

```
void main() {
  // Create a new process
  int ret = fork();
  if (ret == 0) {
    // Child process.
    // Execute “ls -al”
    const char *argv[3];
    argv[0]=“ls”;
    argv[1]=“-al”;
    argv[2] = NULL;
    execvp(argv[0], argv);
    // There was an error
    perror(“execvp”);
    _exit(1);
  }
  else if (ret < 0) {
    // There was an error in fork
    perror(“fork”);
    exit(2);
  }
  else {
    // This is the parent process
    // ret is the pid of the child
    // Wait until the child exits
    waitpid(ret, NULL);
  } // end if
}// end main
```

The execvp() system call

- For lab3 part2 start by creating a new process for each command in the pipeline and making the parent wait for the last command.

```
Command::execute()
{
  int ret;
  for ( int i = 0;
   i < _numberOfSimpleCommands;
      i++ ) {
    ret = fork();
    if (ret == 0) {
      //child
      execvp(sCom[i]->_args[0],
             sCom[i]->_args);
      perror("execvp");
      _exit(1);
    }
    else if (ret < 0) {
      perror("fork");
      return;
    }
    // Parent shell continue
  } // for
  if (!background) {
    // wait for last process
    waitpid(ret, NULL);
  }
}// execute
```

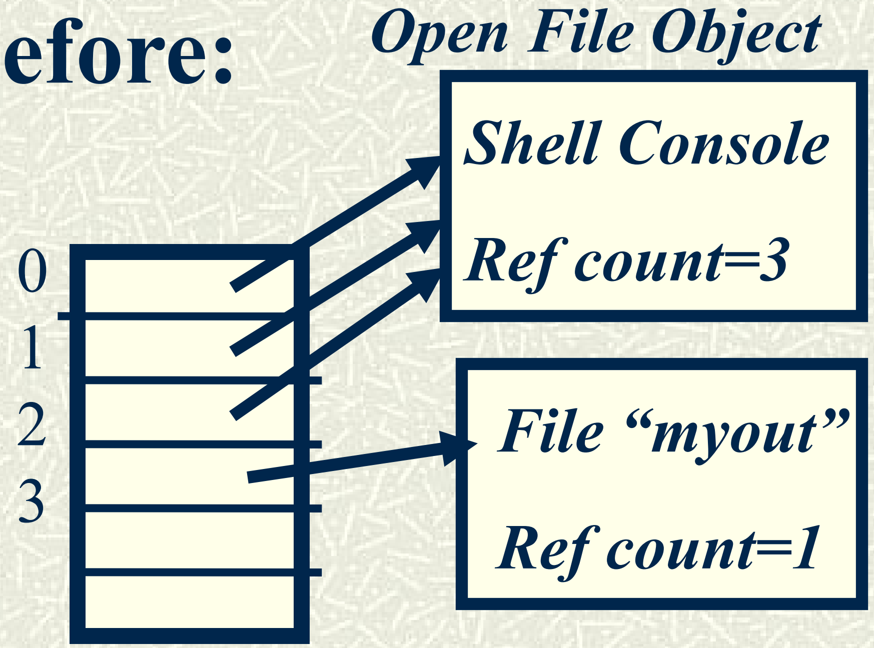
The dup2() system call

- int dup2(fd1, fd2)
  - After dup2(fd1, fd2), fd2 will refer to the same open file object that fd1 refers to.
  - The open file object that fd2 refered to before is closed.
  - The reference counter of the open file object that fd1 refers to is increased.
  - dup2() will be useful to redirect stdin, stdout, and also stderr.

The dup2() system call

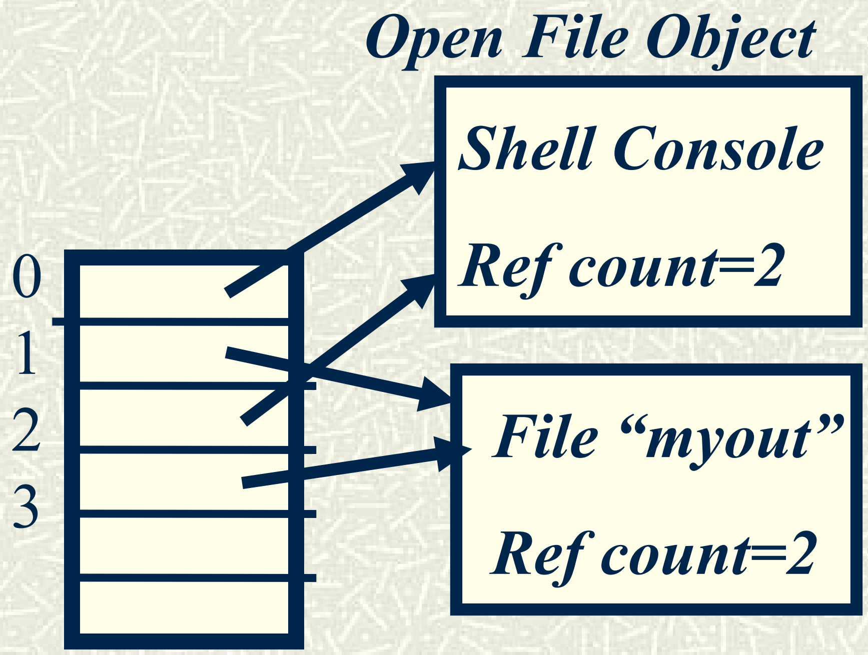
Example: redirecting stdout to file "myfile" previously created.

Before:



The dup2() system call

After dup2(3,1);



•**Now every printf will go to file "myout".**

Example: Redirecting stdout

- A program that redirects stdout to a file myoutput.txt

```
int main(int argc,char**argv)
{
  // Create a new file
  int fd = open(“myoutput.txt”,
      O_CREAT|O_WRONLY|O_TRUNC,
      0664);
  if (fd < 0) {
    perror(“open”);
    exit(1);
  }
  // Redirect stdout to file
  dup2(fd,1);
```

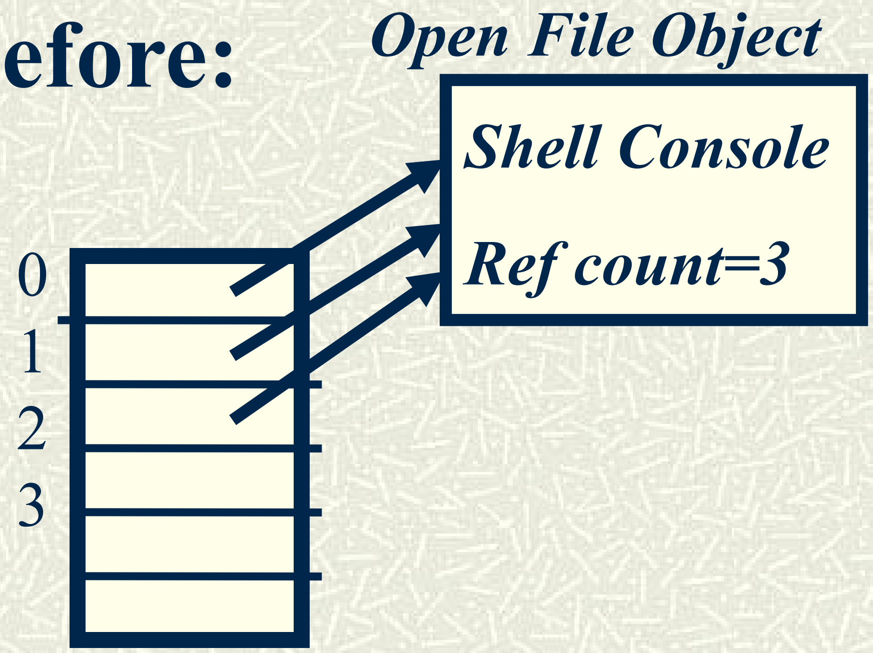
```
  // Now printf that prints
  // to stdout, will write to
  // myoutput.txt
  printf(“Hello world\n”);
}
```

The dup() system call

- fd2=dup(fd1)
  - dup(fd1) will return a new file descriptor that will point to the same file object that fd1 is pointing to.
  - The reference counter of the open file object that fd1 refers to is increased.
  - This will be useful to “save” the stdin, stdout, stderr, so the shell process can restore it after doing the redirection.

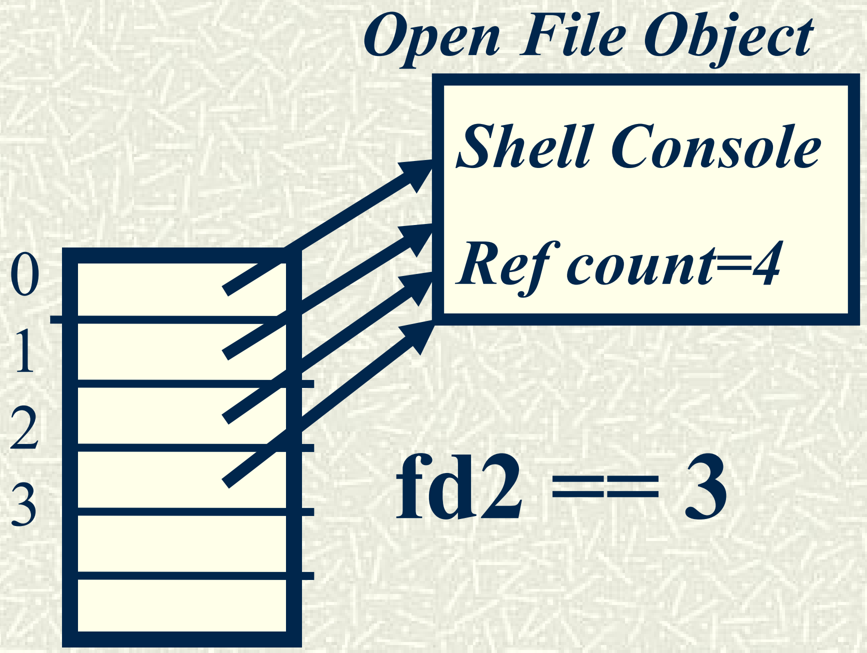
The dup() system call

Before:



The dup() system call

After fd2 = dup(1)



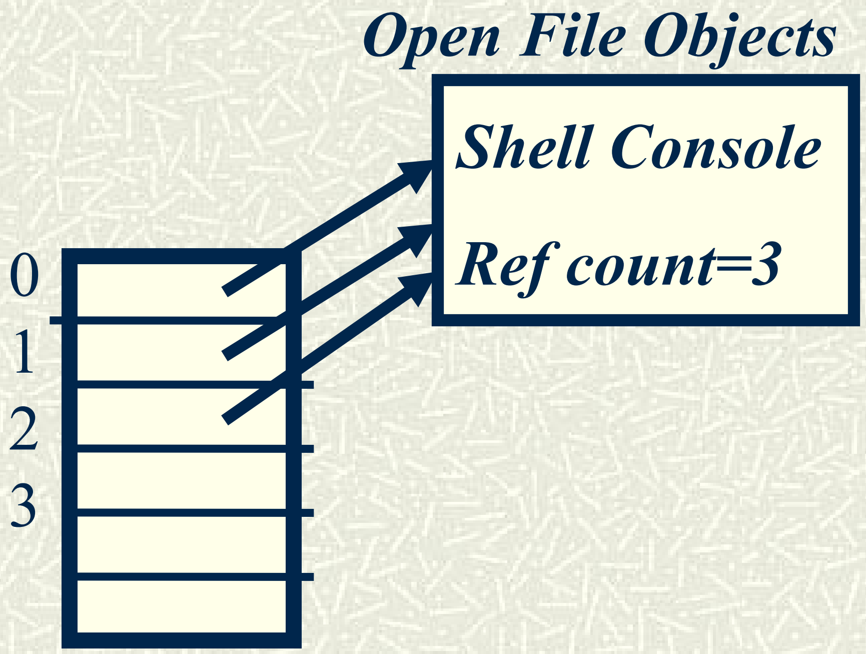
fd2 == 3

The pipe system call

- int pipe(fdpipe[2])
  - fdpipe[2] is an array of int with two elements.
  - After calling pipe, fdpipe will contain two file descriptors that point to two open file objects that are interconnected.
  - What is written into fdpipe[1] can be read from fdpipe[0].
  - In some Unix systems like Solaris pipes are bidirectional but in Linux they are unidirectional.

The pipe system call

Before:



The pipe system call

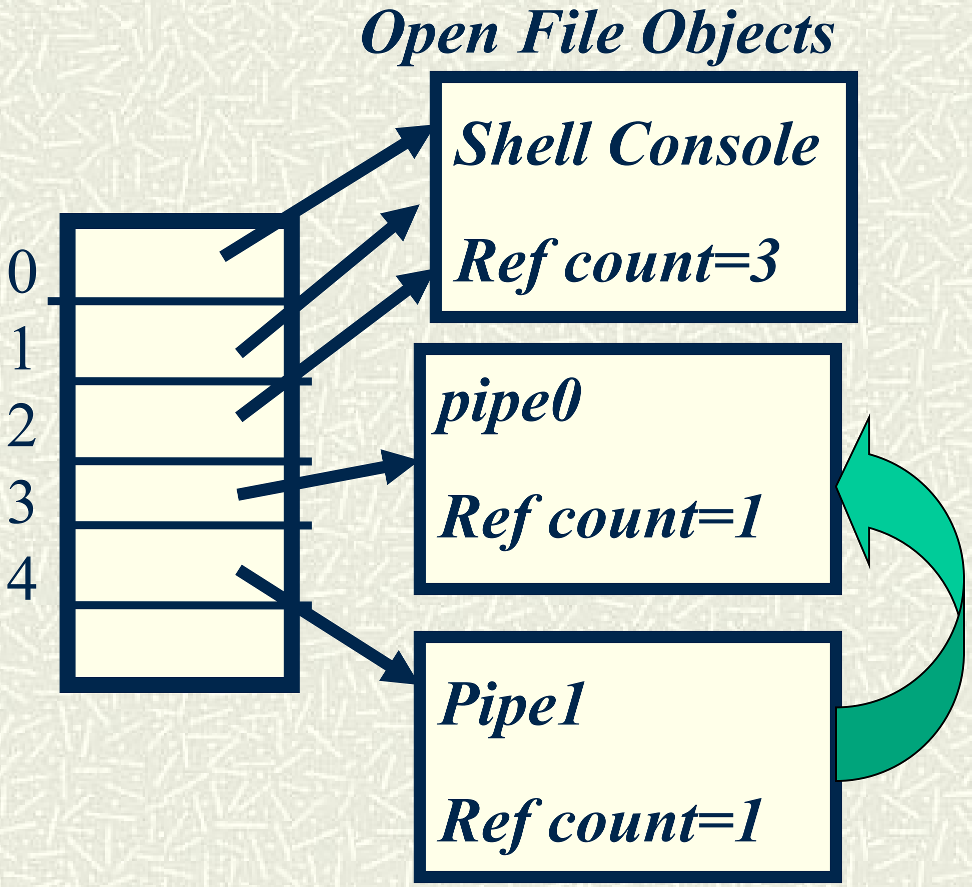
After running:

```
int fdpipe[2];
pipe(fdpipe);
```

fdpipe[0]==3

fdpipe[1]==4

What is written in fdpipe[1] can be read from fdpipe[0].



Example of pipes and redirection

- A program “lsgrep” that runs “ls –al | grep arg1 > arg2”.
- Example: “lsgrep aa myout” lists all files that contain “aa” and puts output in file myout.

```
int main(int argc,char**argv)
{
  if (argc < 3) {
    fprintf(stderr, "usage:”
      “lsgrep arg1 arg2\n");
    exit(1);
  }

  // Strategy: parent does the
  // redirection before fork()
```

```
    //save stdin/stdout
    int tempin = dup(0);
    int tempout = dup(1);
    //create pipe
    int fdpipe[2];
    pipe(fdpipe);

    //redirect stdout for "ls“
    dup2(fdpipe[1],1);
    close(fdpipe[1]);
```

Example of pipes and redirection

```
// fork for "ls"
int ret= fork();
if(ret==0) {
  // close file descriptors
  // as soon as are not
  // needed
  close(fdpipe[0]);
  char * args[3];
  args[0]="ls";
  args[1]="-al";
  args[2]=NULL;
  execvp(args[0], args);
  // error in execvp
  perror("execvp");
  exit(1);
}
```

```
 //redirection for "grep"
 //redirect stdin
 dup2(fdpipe[0], 0);
 close(fdpipe[0]);
 //create outfile
 int fd=open(argv[2],
O_WRONLY|O_CREAT|O_TRUNC,
0600);
 if (fd < 0){
   perror("open");
   exit(1);
 }
  //redirect stdout
 dup2(fd,1);
 close(fd);
```

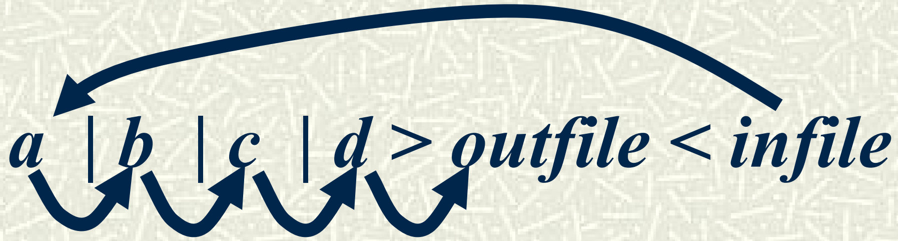
Example of pipes and redirection

```
    // fork for “grep”
    ret= fork();
    if(ret==0) {
      char * args[3];
      args[0]=“grep";
      args[1]=argv[1];
      args[2]=NULL;
      execvp(args[0], args);
      // error in execvp
      perror("execvp");
      _exit(1);
    }
```

```
       // Restore stdin/stdout
        dup2(tempin,0);
        dup2(tempout,1);
        close(tempin);
        close(tmpout)

        // Parent waits for grep
        // process
        waitpid(ret,NULL,0);
        printf(“All done!!\n”);
    } // main
```

Execution Strategy for Your Shell

- Parent process does all the piping and redirection before forking the processes.
- The children will inherit the redirection.
- The parent needs to save input/output and restore it at the end.
- Stderr is the same for all processes



```
a | b | c | d > outfile < infile
```

Execution Strategy for Your Shell

```
execute(){
  //save in/out
  int tmpin=dup(0);
  int tmpout=dup(1);
  //set the initial input
  int fdin;
  if (infile) {
    fdin = open(infile,……);
  }
  else {
    // Use default input
    fdin=dup(tmpin);
  }
```

```
  int ret;
  int fdout;
  for(i=0;i<numsimplecommands;
      i++) {
    //redirect input
    dup2(fdin, 0);
    close(fdin);
    //setup output
    if (i == numsimplecommands-1){
      // Last simple command
      if(outfile){
        fdout=open(outfile,……);
      }
      else {
        // Use default output
        fdout=dup(tmpout);
      }
    }
```

Execution Strategy for Your Shell

```
 else {
  // Not last
  //simple command
  //create pipe
  int fdpipe[2];
  pipe(fdpipe);
  fdout=fdpipe[1];
  fdin=fdpipe[0];
 }// if/else
// Redirect output
dup2(fdout,1);
close(fdout);
```

```
   // Create child process
  ret=fork();
  if(ret==0) {
    execvp(scmd[i].args[0],
          scmd[i].args);
    perror("execvp");
    exit(1);
  }
} //  for
```

Execution Strategy for Your Shell

```
  //restore in/out defaults
  dup2(tmpin,0);
  dup2(tmpout,1);
  close(tmpin);
  close(tmpout);

  if (!background) {
    // Wait for last command
    waitpid(ret, NULL,0 );
  }

} // execute
```

Differences between exit() and _exit()

- exit(int val)
  - It flushes buffers of output streams.
  - Then it exits the current process.
- _exit()
  - It exits immediately without flushing any file buffers.
  - It is recommended to call _exit() in the child process if there is an error after exec.
  - In Solaris, exit() calls lseek to the beginning of the stdin causing problems in the parent.

Notes about Shell Strategy

- The key point is that ***fdin*** is set to be the input for the next command.
- ***fdin*** is a descriptor either of an input file if it is the first command or a ***fdpipe[1]*** if it is not the first command.
- This example only handles pipes and in/out redirection
- You have to redirect stderr for all processes if necessary
- You will need to handle the "append" case

Implementing Wildcards in Shell

- I suggest to implement first the simple case where you expand wildcards in the current directory.
- In shell.y, where arguments are inserted in the table do the expansion.

Implementing Wildcards in Shell

- Before

```
 argument: WORD {
   Command::_currentSimpleCommand->insertArgument($1);
 } ;
```

- After

```
argument: WORD {
   expandWildcardsIfNecessary($1);
 } ;
```

Implementing Wildcards in Shell

```
void expandWildcardsIfNecessary(char * arg)
{
  // Return if arg does not contain '*' or '?'
  if (arg has neither '*' nor '?' (use strchr) ) {
    Command::_currentSimpleCommand->insertArgument(arg);
    return;
  }
```

Implementing Wildcards in Shell

```
// 1. Convert wildcard to regular expression
// Convert “*” -> “.*”
//         “?” -> “.”
//         “.” -> “\.”  and others you need
// Also add ^ at the beginning and $ at the end to match
// the beginning ant the end of the word.
// Allocate enough space for regular expression
char * reg = (char*)malloc(2*strlen(arg)+10);
char * a = arg;
char * r = reg;
*r = ‘^’; r++; // match beginning of line
while (*a) {
  if (*a == ‘*’) { *r=‘.’; r++; *r=‘*’; r++; }
  else if (*a == ‘?’) { *r=‘.’ r++;}
  else if (*a == ‘.’) { *r=‘\\’; r++; *r=‘.’; r++;}
  else { *r=*a; r++;}
  a++;
}
*r=‘$’; r++; *r=0;// match end of line and add null char
```

Implementing Wildcards in Shell

```
// 2. compile regular expression. See lab3-src/regular.cc
char * expbuf = regcomp( reg, … );
if (expbuf==NULL) {
  perror("compile");
  return;
}

// 3. List directory and add as arguments the entries
// that match the regular expression
DIR * dir = opendir(".");
if (dir == NULL) {
  perror("opendir");
  return;
}
```

Implementing Wildcards in Shell

```
  struct dirent * ent;
  while ( (ent = readdir(dir))!= NULL) {
    // Check if name matches
    if (regexec(ent->d_name, expbuf ) ==0 ) {
      // Add argument
      Command::_currentSimpleCommand->
          insertArgument(strdup(ent->d_name));
    }
  }
  closedir(dir);
}
```

Note: This code is not complete and contains errors. The purpose of this code is only to guide your implementation.

Sorting Directory Entries

- Shells like /bin/sh sort the entries matched by a wildcard.
- For example “echo *” will list all the entries in the current directory sorted.
- You will have to modify the wildcard matching as follows:

Sorting Directory Entries

```
struct dirent * ent;
int maxEntries = 20;
int nEntries = 0;
char ** array = (char**) malloc(maxEntries*sizeof(char*));
while ( (ent = readdir(dir))!= NULL) {
  // Check if name matches
  if (regexec(ent->d_name, expbuf) ) {
    if (nEntries == maxEntries) {
      maxEntries *=2;
      array = realloc(array, maxEntries*sizeof(char*));
      assert(array!=NULL);
    }
    array[nEntries]= strdup(ent->d_name);
    nEntries++;
  }
}
```

Sorting Directory Entries

```
 closedir(dir);
 sortArrayStrings(array, nEntries);
// Add arguments
for (int i = 0; i < nEntries; i++) {
    Command::_currentSimpleCommand->
      insertArgument(array[i]));
}
free(array);
```

Wildcards and Hidden Files

- In UNIX invisible files start with "." like .login, .bashrc etc.
- In these files that start with ".", the "." should not be matched with a wildcard.
- For example "echo *" will not display "." and "..".
- To do this, you will add a filename that starts with "." only if the wildcard has a "." at the beginning.

Wildcards and Hidden Files

```
if (regexec (…) ) {
  if (ent->d_name[0] == ‘.’) {
    if (arg[0] == ‘.’)
      add filename to arguments;
    }
  }
  else {
    add ent->d_name to arguments
  }
}
```

Subdirectory Wildcards

- Wildcards also match directories inside a path:

  Eg. echo /p*/*a/b*/aa*

- To match subdirectories you need to match component by component

Subdirectory Wildcards

$$\begin{tikzcd}
 & & \text{/usr/lib/a*} \arrow[r] \arrow[dr] & \text{/usr/lib/aa} \\
 & \text{/usr/l*/a*} \arrow[ur] \arrow[dr] & & \text{/usr/lib/abb} \\
 & & \text{/usr/local/a*} \arrow[r] \arrow[dr] & \text{/usr/local/axz} \\
 & & & \text{/usr/local/a12} \\
 & & \text{/u/loc/a*} \arrow[r] \arrow[dr] & \text{/u/loc/aq} \\
\text{/u*/l*/a*} \arrow[uuuur] \arrow[r] \arrow[ddddr] & \text{/u/l*/a*} \arrow[ur] \arrow[dr] & & \text{/u/loc/ar} \\
 & & \text{/u/lll/a*} \arrow[r] \arrow[dr] & \text{/u/lll/apple} \\
 & & & \text{/u/lll/atp} \\
 & & \text{/unix/l34/a*} \arrow[r] \arrow[dr] & \text{/unix/l34/abcd} \\
 & \text{/unix/l*/a*} \arrow[ur] \arrow[dr] & & \text{/unix/l34/a45} \\
 & & \text{/unix/lll/a*} \arrow[r] \arrow[dr] & \text{/unix/lll/ab} \\
 & & & \text{/unix/lll/ap3}
\end{tikzcd}$$

Subdirectory Wildcards

- Strategy:
  - Write a function

    `expandWildcard(prefix, suffix)` where

    - **`prefix`**- The path that has been expanded already. It should not have wildcards.
    - **`suffix`** – The remaining part of the path that has not been expanded yet. It may have wildcards.

$$\underbrace{\texttt{/usr}}_{\text{preffix}}\underbrace{\texttt{/l*/a*}}_{\text{suffix}}$$

  - The prefix will be inserted as argument when the suffix is empty
  - *`expandWildcard(prefix, suffix)`* is initially invoked with an empty prefix and the wildcard in suffix.

Subdirectory Wildcards

```
#define MAXFILENAME 1024
void expandWildcard(char * prefix, char *suffix) {
  if (suffix[0]== 0) {
       // suffix is empty. Put prefix in argument.
       …->insertArgument(strdup(prefix));
       return;
  }
  // Obtain the next component in the suffix
  // Also advance suffix.
  char * s = strchr(suffix, '/');
  char component[MAXFILENAME];
  if (s!=NULL){ // Copy up to the first "/"
    strncpy(component,suffix, s-suffix);
    suffix = s + 1;
  }
  else { // Last part of path. Copy whole thing.
    strcpy(component, suffix);
    suffix = suffix + strlen(suffix);
  }
```

Subdirectory Wildcards

```
// Now we need to expand the component
char newPrefix[MAXFILENAME];
if ( component does not have '*' or '?') {
  // component does not have wildcards
  sprintf(newPrefix,"%s/%s", prefix, component);
  expandWildcard(newPrefix, suffix);
  return;
}
// Component has wildcards
// Convert component to regular expression
char * expbuf = compile(…)
char * dir;
// If prefix is empty then list current directory
if (prefix is empty) dir =".”; else dir=prefix;
DIR * d=opendir(dir);
if (d==NULL) return;
```

Subdirectory Wildcards

```
  // Now we need to check what entries match
  while ((ent = readdir(d))!= NULL) {
     // Check if name matches
     if (advance(ent->d_name, expbuf) ) {
        // Entry matches. Add name of entry
        // that matches to the prefix and
        // call expandWildcard(..) recursively
        sprintf(newPrefix,"%s/%s", prefix, ent->d_name);
        expandWildcard(newPrefix,suffix);
     }
  }
  close(d);
}// expandWildcard
```

Executing built-in functions

- All built-in functions except printenv are executed by the parent process.
- Why? we want setenv, cd etc to modify the state of the parent. If they are executed by the child, the changes will go away when the child exits.

Executing printenv

- Execute printenv in the child process and exit. In this way the output of printenv could be redirected.

```
ret = fork();
if (ret == 0) {
  if (!strcmp(argument[0],printenv)) {
    char **p=environ;
    while (*p!=NULL) {
      printf("%s",*p);
      p++;
    }
    exit(0);
  }
  …
  execvp(…);
  …
```

Signals and the Lex Scanner

- The scanner implemented by lex calls getc() to get the next char for stdin.
- getc() calls the system call read().
- System calls that block will return if a signal is received. The errno will be EINTR.
- Ctrl-c generates the SIGINT signal. A child that exits will generate SIGCHLD.
- In any of these signals getc(), will return –1 (EOF) and it will exit.

Signals and the Lex Scanner

- To prevent a system call to be interrupted, use sigaction when setting the signal handler and also set the flag SA_RESTART.
- SA_RESTART will retry the system call if interrupted by a signal

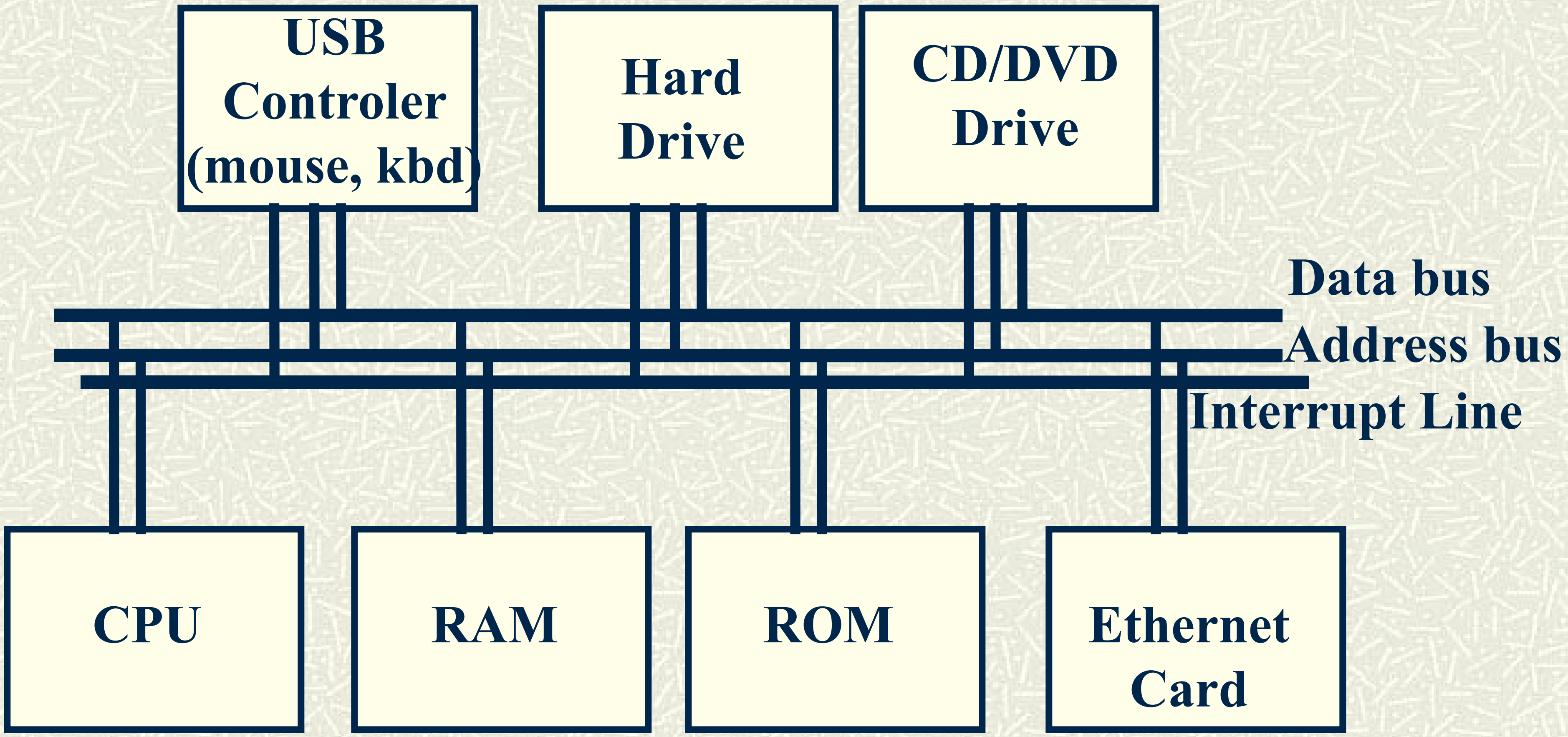
```
struct sigaction signalAction;
signalAction.sa_handler = sigIntHandler;
sigemptyset(&signalAction.sa_mask);
signalAction.sa_flags = SA_RESTART;
int error =
  sigaction(SIGINT, &signalAction, NULL );
if ( error ) {
  perror( "sigaction" );
  exit( -1 );
}
```

User Mode, Kernel Mode, Iterrupts and System Calls

Computer Architecture Review

- Most modern computers use the ***Von Newman Architecture*** where both programs and data are stored in RAM.
- A computer has an ***address bus*** and a ***data bus*** that are used to transfer data from/to the CPU, RAM, ROM, and the devices.
- The CPU, RAM, ROM, and all devices are attached to this bus.

Computer Architecture Review



Kernel and User Mode

- Kernel Mode
  - When the CPU runs in this mode:
    - It can run any instruction in the CPU
    - It can modify any location in memory
    - It can access and modify any register in the CPU and any device.
    - There is full control of the computer.
  - The OS Services run in kernel mode.

Kernel and User Mode

- **User Mode**
  - When the CPU runs in this mode:
    - The CPU can use a limited set of instructions
    - The CPU can only modify only the sections of memory assigned to the process running the program.
    - The CPU can access only a subset of registers in the CPU and it cannot access registers in devices.
    - There is a limited access to the resources of the computer.
  - The user programs run in user mode

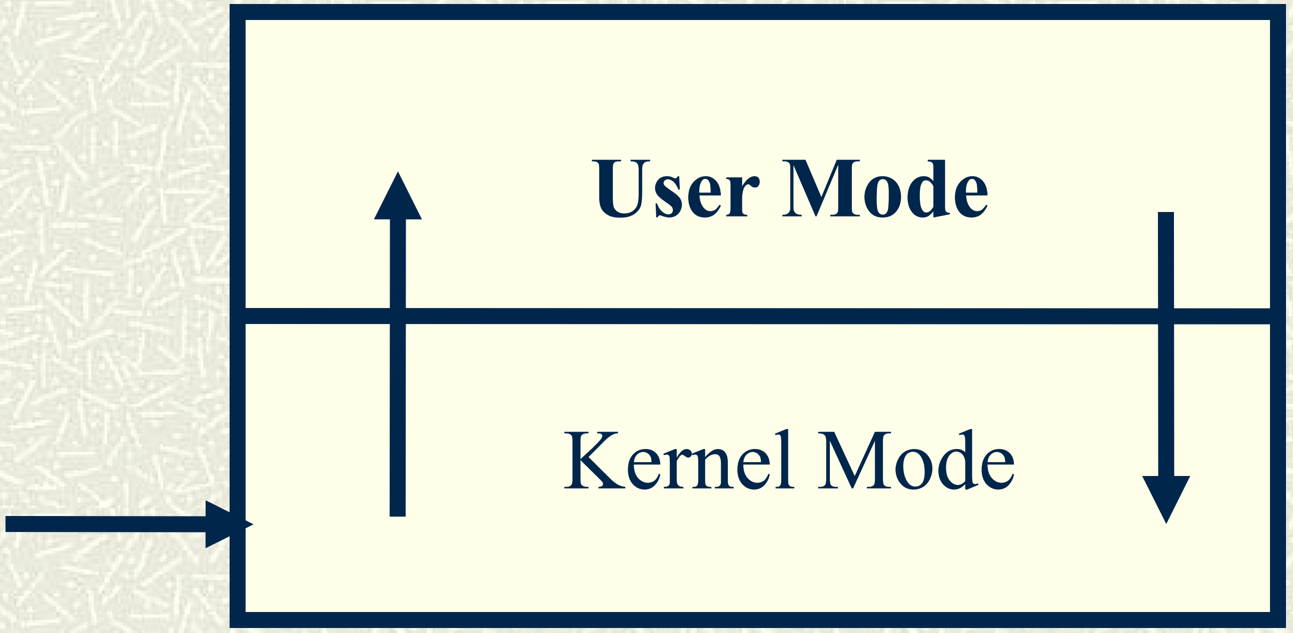
Kernel and User Mode

- When the OS boots, it starts in kernel mode.
- In kernel mode the OS sets up all the interrupt vectors and initializes all the devices.
- Then it starts the first process and switches to user mode.
- In user mode it runs all the background system processes (daemons or services).
- Then it runs the user shell or windows manager.

Kernel and User Mode

- User programs run in user mode.
- The programs switch to kernel mode to request OS services (system calls)
- Also user programs switch to kernel mode when an interrupt arrives.
- They switch back to user mode when interrupt returns.
- The interrupts are executed in kernel mode.
- The interrupt vector can be modified only in kernel mode.
- Most of the CPU time is spent in User mode

Kernel and User Mode



Kernel and User Mode

- Separation of user/kernel mode is used for:
  - Security: The OS calls in kernel mode make sure that the user has enough privileges to run that call.
  - Robustness: If a process that tries to write to an invalid memory location, the OS will kill the program, but the OS continues to run. A crash in the process will not crash the OS. > A bug in user mode causes program to crash, OS runs. A bug in kernel mode may cause OS and system to crash.
  - Fairness: OS calls in kernel mode to enforce fair access.

Interrupts

- An interrupt is an event that requires immediate attention. In hardware, a device sets the interrupt line to high.
- When an interrupt is received, the CPU will stop whatever it is doing and it will jump to to the 'interrupt handler' that handles that specific interrupt.
- After executing the handler, it will return to the same place where the interrupt happened and the program continues. Examples:
  - move mouse
  - type key
  - ethernet packet

Steps of Servicing an Interrupt

1. The CPU saves the Program Counter and registers in execution stack
2. CPU looks up the corresponding interrupt handler in the interrupt vector.
3. CPU jumps to interrupt handler and run it.
4. CPU restores the registers and return back to the place in the program that was interrupted. The program continues execution as if nothing happened.
5. In some cases it retries the instruction the instruction that was interrupted (E.g. Virtual memory page fault handlers).

Running with Interrupts

- Interrupts allow CPU and device to run in parallel without waiting for each other.

1. OS Requests Device Operation (E.g.Write to disk)

2. Device Runs Operation

2. OS does other things in parallel with device.

3. When Operation is complete interrupt OS

4. OS services interrupt and continues

Poling

- Alternatively, the OS may decide not use interrupts for some devices and wait in a busy loop until completion.

```
OS requests Device operation
While request is not complete
  do nothing;
Continue execution.
```

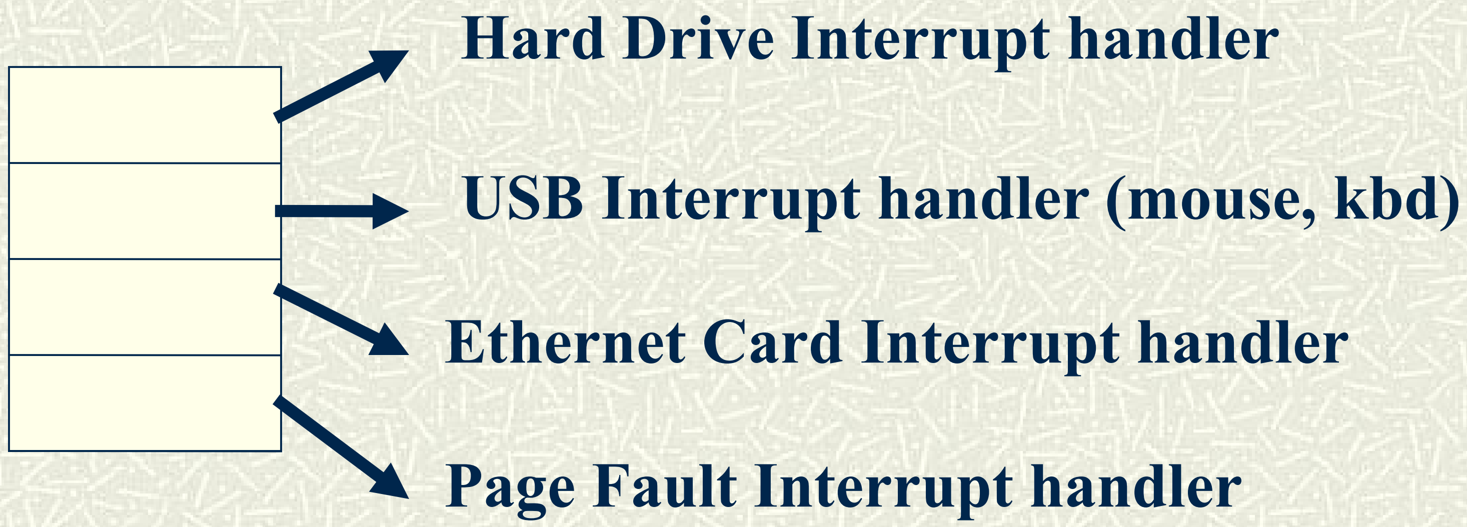
- This type of processing is called "poling" or "busy waiting" and wastes a lot of CPU cycles.
- Poling is used for example to print debug messages in the kernel (kprintf). We want to make sure that the debug message is printed to before continuing the execution of the OS.

Synchronous vs. Asynchronous

- ***Poling*** is also called ***Synchronous Processing*** since the execution of the device is synchronized with the program.
- An ***interrupt*** is also called ***Asynchronous Processing*** because the execution of the device is not synchronized with the execution of the program. Both device and CPU run in parallel.

Interrupt Vector

- It is an array of pointers that point to the different interrupt handlers of the different types of interrupts.


Hard Drive Interrupt handler
USB Interrupt handler (mouse, kbd)
Ethernet Card Interrupt handler
Page Fault Interrupt handler

Interrupts and Kernel Mode

- Interrupts run in kernel mode. Why?
  - An interrupt handler must read device/CPU registers and execute instructions only available in kernel mode.
- Interrupt vector can be modified only in kernel mode (security)
- Interrupt vector initialized during boot time; modified when drivers added to system

Types of Interrupts

1. **Device Interrupts** generated by Devices when a request is complete or an event that requires CPU attention happens.
   - The mouse is moved
   - A key is typed
   - An Ethernet packet arrives.
   - The hard drive has completed a read/write operation.
   - A CD has been inserted in the CD drive.

Types of Interrupts

2. **Math exceptions** generated by the CPU when there is a math error.
   - Divide by zero
3. ***Page Faults*** generated by the MMU (Memory Management Unit) that converts Virtual memory addresses to physical memory addresses
   - Invalid address: interrupt prompts a SEGV signal to the process
   - Access to a valid address but there is not page in memory. This causes the CPU to load the page from disk
   - Invalid permission (I.e. trying to write on a read only page) causes a SEGV signal to the process.

Types of Interrupts

4. **Software Interrupt** generated by software with a special assembly instruction. This is how a program running in user mode requests operating systems services.

System Calls

- System Calls is the way user programs request services from the OS
- **System calls** use **Software Interrupts**
- Examples of system calls are:
  - `open(filename, mode)`
  - `read(file, buffer, size)`
  - `write(file, buffer, size)`
  - `fork()`
  - `execve(cmd, args);`
- System calls is the API of the OS from the user program’s point of view. See /usr/include/sys/syscall.h

Why do we use Software Interrupts for syscalls instead of function calls?

- Software Interrupts will switch into kernel mode
- OS services need to run in kernel mode because:
  - They need privileged instructions
  - Accessing devices and kernel data structures
  - They need to enforce the security in kernel mode.

System Calls

- Only operations that need to be executed by the OS in kernel mode are part of the system calls.
- Function like sin(x), cos(x) are not system calls.
- Some functions like printf(s) run mainly in user mode but eventually call ***`write()`*** when for example the buffer is full and needs to be flushed.
- Also ***malloc(size)*** will run mostly in user mode but eventually it will call ***sbrk()*** to extend the heap.

System Calls

- Libc (the C library) provides wrappers for the system calls that eventually generate the system calls.

User Mode:

```
int open(fname, mode) {
   return syscall(SYS_open,
   fname, mode);
}
int syscall(syscall_num, …)
{
   asm(INT);
}
```

Software Interrupt

Kernel Mode:

```
Syscall interrupt handler:
Read:…
Write:…
open:
   - Get file name and mode
   - Check if file exists
   - Verify permissions of
     file against mode.
   - Ask disk to Perform operation.
     Make process wait until
     operation is complete
   - return fd (file
     descriptor)
```

System Calls

- The software interrupt handler for system calls has entries for all system calls.
- The handler checks that the arguments are valid and that the operation can be executed.
- The arguments of the syscall are checked to enforce the security and protections.

Syscall Security Enforcement

- For example, for the open syscall the following is checked in the syscall software interrupt handler:

  open(filename, mode)

  - If file does not exist return error
  - If permissions of file do not agree with the mode the file will be opened, return error. Consider also who the owner of the file is and the owner of the process calling open.
  - If all checks pass, open file and return file handler.

Syscall details

- Te list of all system calls can be found in /usr/include/sys/syscall.h

```
#define SYS_exit        1
#define SYS_fork        2
#define SYS_read        3
#define SYS_write       4
#define SYS_open        5
#define SYS_close       6
#define SYS_wait        7
#define SYS_creat       8
#define SYS_link        9
#define SYS_unlink      10
#define SYS_exec        11
…
```

Syscall Error reporting

- When an error in a system call occurrs, the OS sets a global variable called “errno” defined in libc.so with the number of the error that gives the reason for failure.
- The list of all the errors can be found in /usr/include/sys/errno.h

```
#define EPERM    1         /* Not super-user     */
#define ENOENT   2         /* No such file or directory */
#define ESRCH    3         /* No such process */
#define EINTR    4         /* interrupted system call  */
#define EIO      5         /* I/O error  */
#define ENXIO    6         /* No such device or address */
```

- You can print the corresponding error message to stderr using `perror(s);` where s is a string prepended to the message.

System Calls and Interrupts Example

1. The user program calls the `write(fd, buff, n)` system call to write to disk.
2. The write wrapper in libc generates a software interrupt for the system call.
3. The OS in the interrupt handler checks the arguments. It verifies that fd is a file descriptor for a file opened in write mode. And also that [buff, buff+n-1] is a valid memory range. If any of the checks fail write return -1 and sets errno to the error value.

System Calls and Interrupts Example

4. The OS tells the hard drive to write the buffer in [buff, buff+n] to disk to the file specified by fd.
5. The OS puts the current process in wait state until the disk operation is complete. Meanwhile, the OS switches to another process.
6. The Disk completes the write operation and generates an interrupt.
7. The interrupt handler puts the process calling `write` into ready state so this process will be scheduled by the OS in the next chance.

Strace command

- It is a tool that prints the system calls that a program calls.

```
hello.c
main()
{
   printf("Hello cs252\n");
   write(1, "Hello again\n", 12);
}
$ gcc –o hello hello.c
$ strace ./hello
execve("./hello", ["./hello"], [/* 40 vars */]) = 0
brk(0)                                  = 0x2520000
mmap(NULL, 4096, PROT_READ|PROT_WRITE,
     MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f2e20382000
…
write(1, "Hello cs252\n", 12Hello cs252)          = 12
write(1, "Hello again\n", 12Hello again)          = 12
exit_group(12)                          = ?
```

Processes

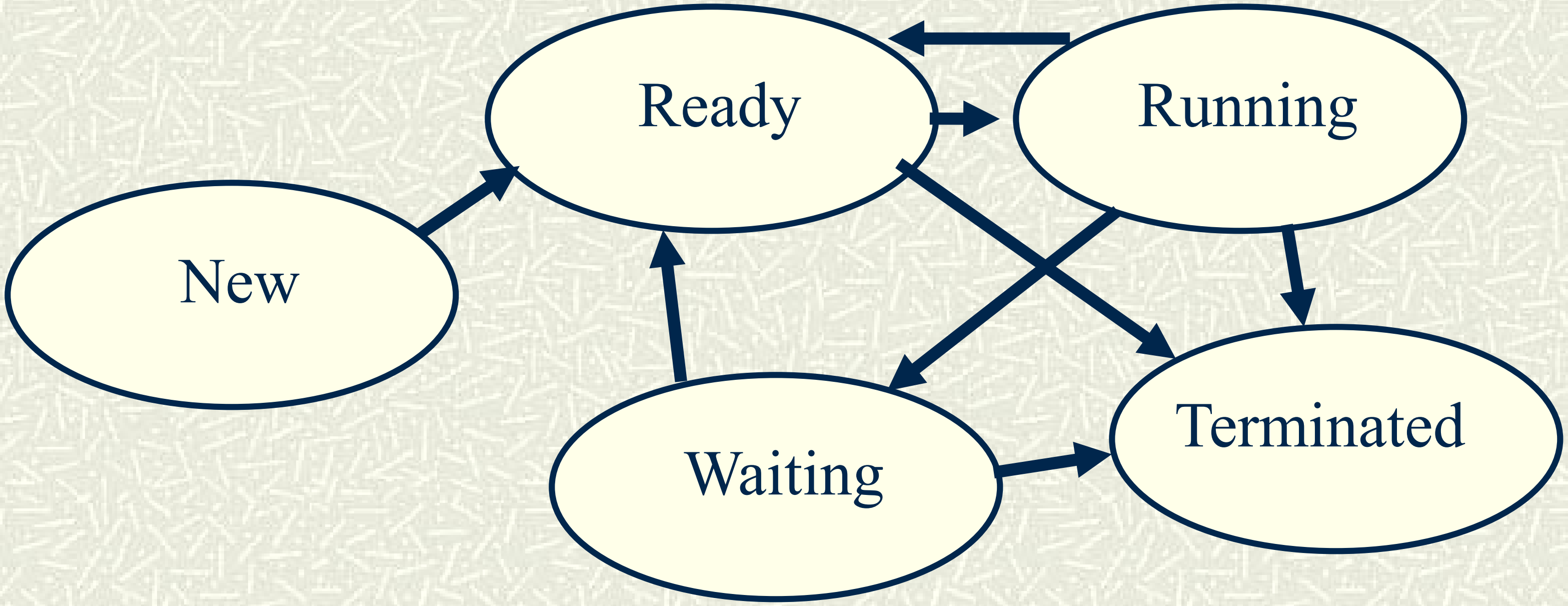
Processes

- A process is a program in execution
- A program may have multiple processes running the same program. E.g. csh running for multiple users or multiple times for the same user.
- Each process will be a different instance of the same program.
- All processes have a parent process except for the first process (init process 0).
- To list processes use
  - `ps` - List processes of the current shell session
  - `ps -u <your-login>` - List processes owned by you
  - `ps -e` - List all processes of the system

Example of ps command

```
brastius 636 % ps -e
   PID TTY      TIME CMD
     0 ?        0:17 sched
     1 ?        0:08 init
     2 ?        0:00 pageout
     3 ?      112:48 fsflush
   317 ?        0:00 xdm
   218 ?        0:01 cron
   248 ?        0:00 sendmail
    57 ?        0:00 sysevent
    72 ?        0:00 picld
   140 ?        0:20 in.route
   153 ?        2:17 sshd
   158 ?        0:43 rpcbind
```

States of a Process



States of a Process

- **New**
  - Process is being initialized
- **Ready**
  - The process is a candidate to run in the CPU but there is another process currently running
- **Running**
  - The process is currently using the CPU.
  - The number of running processes is less than or equal to the number of CPUs
- **Waiting**
  - Process is waiting for an event
- **Terminated**
  - Process is exiting

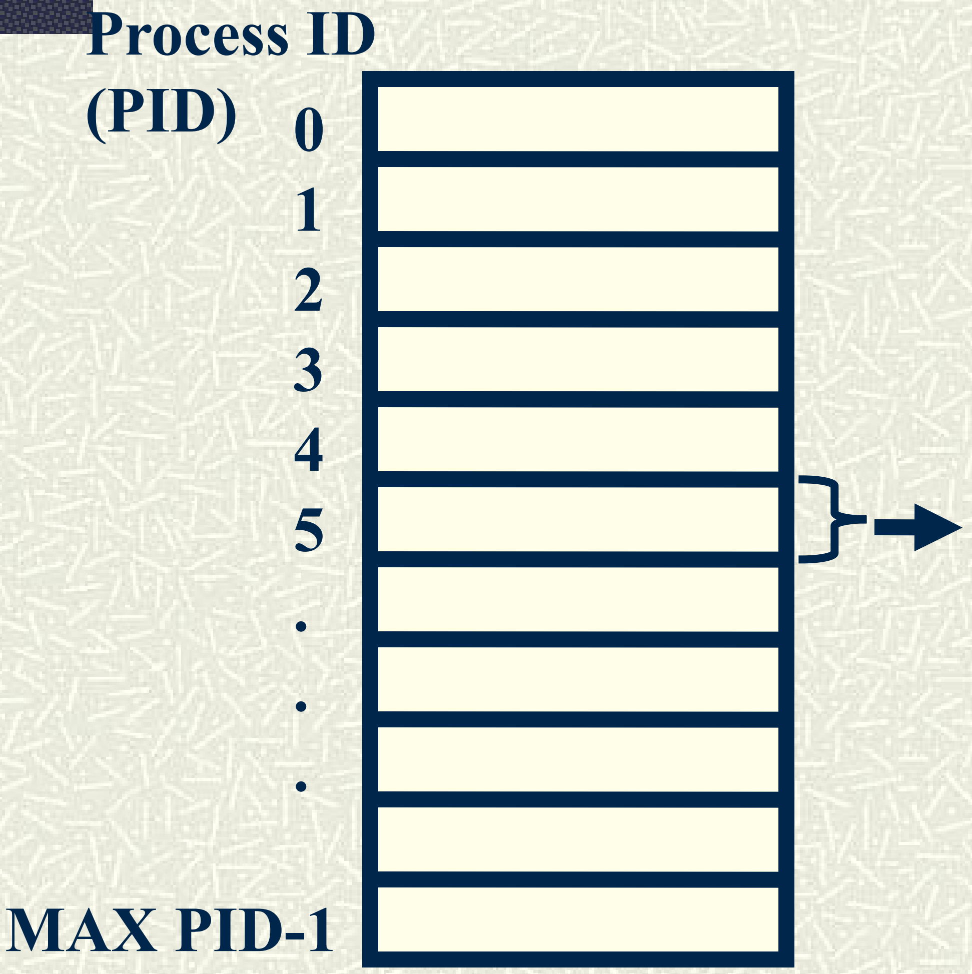
I/O and CPU Bound Process

- ***I/O bound processes***
  - Processes that are most of the time waiting for an event: mouse event, keyboard, Ethernet packet arrives etc.
  - This type of process is in ready/running state only for a short period of time: E.g. update mouse cursor after mouse movement, show a character after typing a key.
  - Most processes are in ***waiting state***.
- ***CPU bound process***:
  - These processes need the CPU for long periods of time: Scientific/Numerical Applications, Compiler/Optimizer, Renderer, Image processing
  - These processes are mostly in ***ready or running state***
- ***Most applications are I/O bound***

Process Table

- Each process will be represented with an entry in the process table.
- The Process Table is one of the most important data structures in the kernel.
- The maximum number of entries in the process table determines the maximum number of processes in the system and it is set at boot time.

Process Table



Each Entry Includes:

-PID: Index in process table
-Command and args
-Environment Vars
-Current Dir
-Owner (User ID)
-Stdin/Stdout/Stderr
-List of memory mappings used by process
-PC (Program Counter)
-Registers
-Stack
-List of Open Files
-State (Wait, Ready, Running etc.)

Process Table

- Each process table entry contains enough information to restart (put in *running* state) a process that is *waiting* or in *ready* state.
- A process may have more than one thread.
- There is a stack, pc and a set of registers and state in the process table entry for each thread that the process has.

Process Scheduling

- From the user's point of view, an OS allows running multiple processes simultaneously.
- In reality, the OS runs ***one process after another*** to give the illusion that multiple processes run simultaneously.
- The ***Process Scheduler*** is the OS subsystem that runs one process after the other and decides what process to run next.
- ***A Context Switch*** is the procedure used by the OS to switch from one process to another

Process Scheduling

- Steps of a Context Switch
  - Save current running process in process table
  - Load next ready process from process table and put registers and PC in the CPU.
- Context Switches may happen as a result of:
  - A process needs to go to wait state, and therefore, another process can be scheduled.
  - A process yields (gives up) the CPU so another process can run.
  - Timer interrupt (Only for Preemptive Scheduling)

Types of Scheduling

- There are two types of scheduling:
  - Non Preemptive Scheduling
  - Preemptive Scheduling

Non Preemptive Scheduling

- In ***Non Preemptive Scheduling*** a **context switch** (switch from one process to another) happens only when the **running process goes to a waiting state** or when the process gives up the CPU voluntarily.
- This is a very primitive type of scheduling. It is also called ***cooperative scheduling***.
- It was used in Windows 3.1 and initial versions of MacOS.
- The main problem of Non Preemptive Scheduling is that a misbehaved process that loops forever may hold the CPU and prevent other processes from running.

Preemptive Scheduling

- In ***Preemptive Scheduling*** a context switch happens periodically (**every 1/100sec** or quantum time) as a result of a timer interrupt.
- A **timer interrupt** will cause a **context switch**, that is, the running process to go to ready state and the process that has been the longest in ready state will go to running state.
- Preemptive scheduling is implemented in UNIX, and Windows 95 and above.

Advantages/Disadvantages of Non Preemptive Scheduling

- **Advantages of Non Preemptive Scheduling**:
  - More control on how the CPU is used.
  - Simple to implement.
- **Advantages of Preemptive scheduling:**
  - More robust, one process cannot monopolize the CPU
  - Fairness. The OS makes sure that CPU usage is the same by all running process.

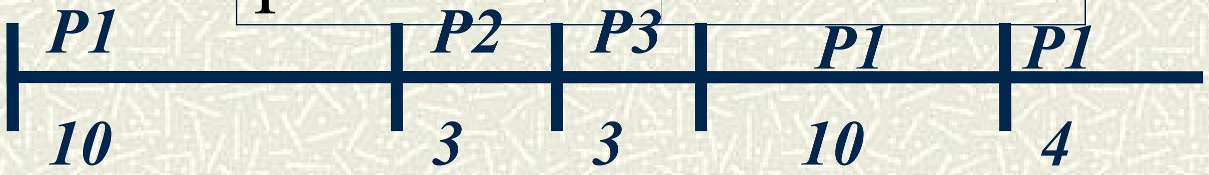
Scheduling Policies for Preemptive Scheduling

- Two policies:
  - Round Robin
  - Multilevel Feedback-Queue Scheduling
- Round Robin
  - Ready processes are in a queue.
  - Every time that a time quantum expires, a process is dequeued from the front of the queue and put in running state
  - The previous running process is added to the end of the queue.

Round Robin

- Round Robin (cont.) (T=10ms)

Process	Burst Time
p1	24ms
p2	3ms
p3	3ms

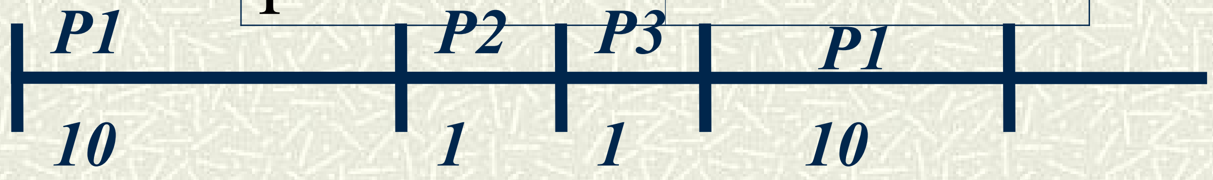


Average completion time = (13+16+30)/3=19.6ms

Quantum Length Analysis

- What is the impact of quantum length? Assume T=10ms.

Process	Burst Time
p1	20ms
p2	1ms
p3	1ms



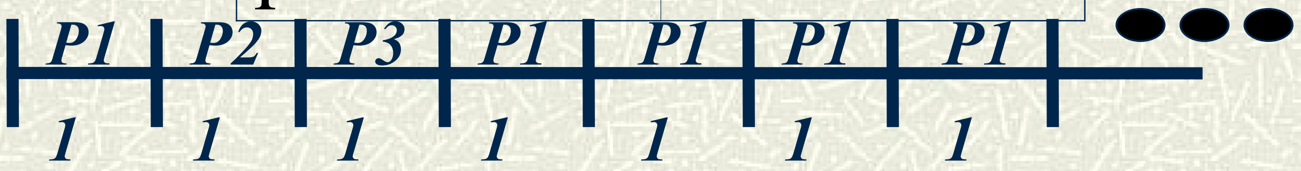
Average completion time = (11+12+22)/3=15ms

Quantum Length Analysis

- What if we choose a smaller quantum length?Assume T=1ms.

T=1ms

Process	Burst Time
p1	20ms
p2	1ms
p3	1ms



Average completion time = (2+3+22)/3=9ms

Quantum Length Analysis

- *The shorter the quantum, the shorter the average completion time.*
- Based on this we could make the quantum time very small to achieve faster response time. What is the catch?
- We are not considering that context switch takes time.
- We have to consider the context switch overhead.

Context Switch Overhead

- The context switch overhead is the time it takes to do a context switch as a portion of the quantum time.

  Xoverhd% = 100*X/T   where

  Xoverhd% = Context Switch Overhead

  X = Context Switch Time

  T = Quantum Time.

- Assume X=.1ms.

  For T=10ms Ovhd=100*.1/10=1%

  For T=2ms Ovhd=100*.1/2=5%

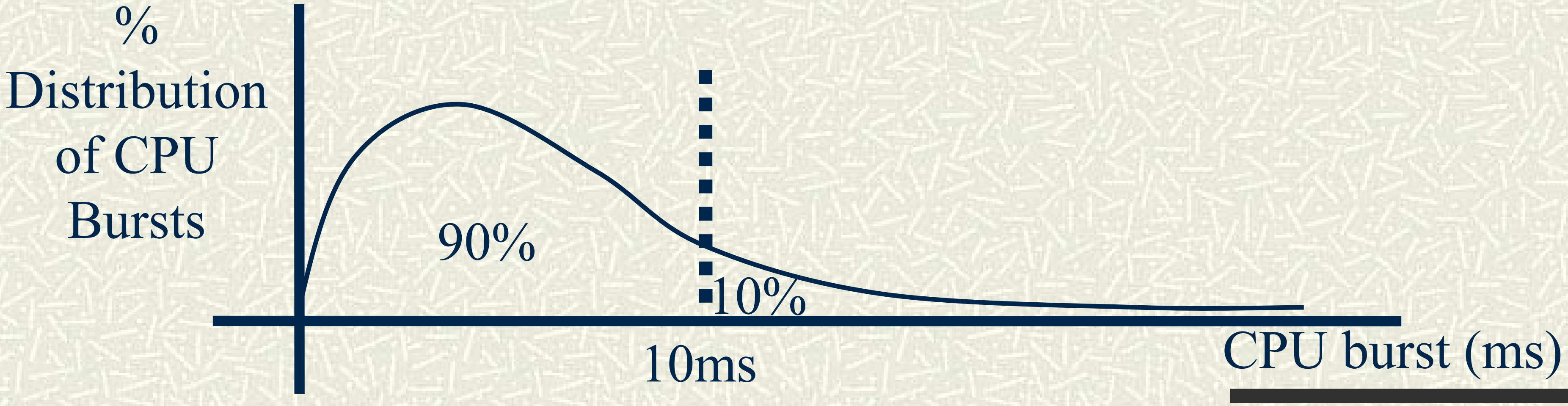
  For T=.2ms Ovhd=100*.1/.2=50%

Context Switch Overhead

- Conclusions:
  - ***The smaller the quantum, the faster the response time (small average completion time) but the larger the context switch overhead.***
  - ***The larger the quantum the smaller the context switch overhead but the larger the response (large average completion time).***
- The standard quantum time is 1/100sec = 10ms that is a compromise between response time and context switch overhead. This may change in the future with faster processors.

CPU Burst Distribution

- Processes are in waiting state most of the time except for the times they need the CPU, that is usually for short periods of time.
- These shorts periods of time the processes need the CPU are called ***CPU bursts***.



CPU Burst Distribution

- 90% of CPU bursts are smaller than 10ms.
- By choosing 10ms to be the quantum length, we make sure that 90% of the CPU burst run until completion without being preempted.
- This reduces the context switch overhead and reduces the completion time and makes the response time faster.

How to Choose the Size of a Quantum

- ***Make the quantum small enough to make the response time smaller (faster).***
- ***Make the quantum large enough to have an acceptable context switch overhead.***
- ***Make the quantum large enough so most CPU bursts are able to finish without being preempted.***

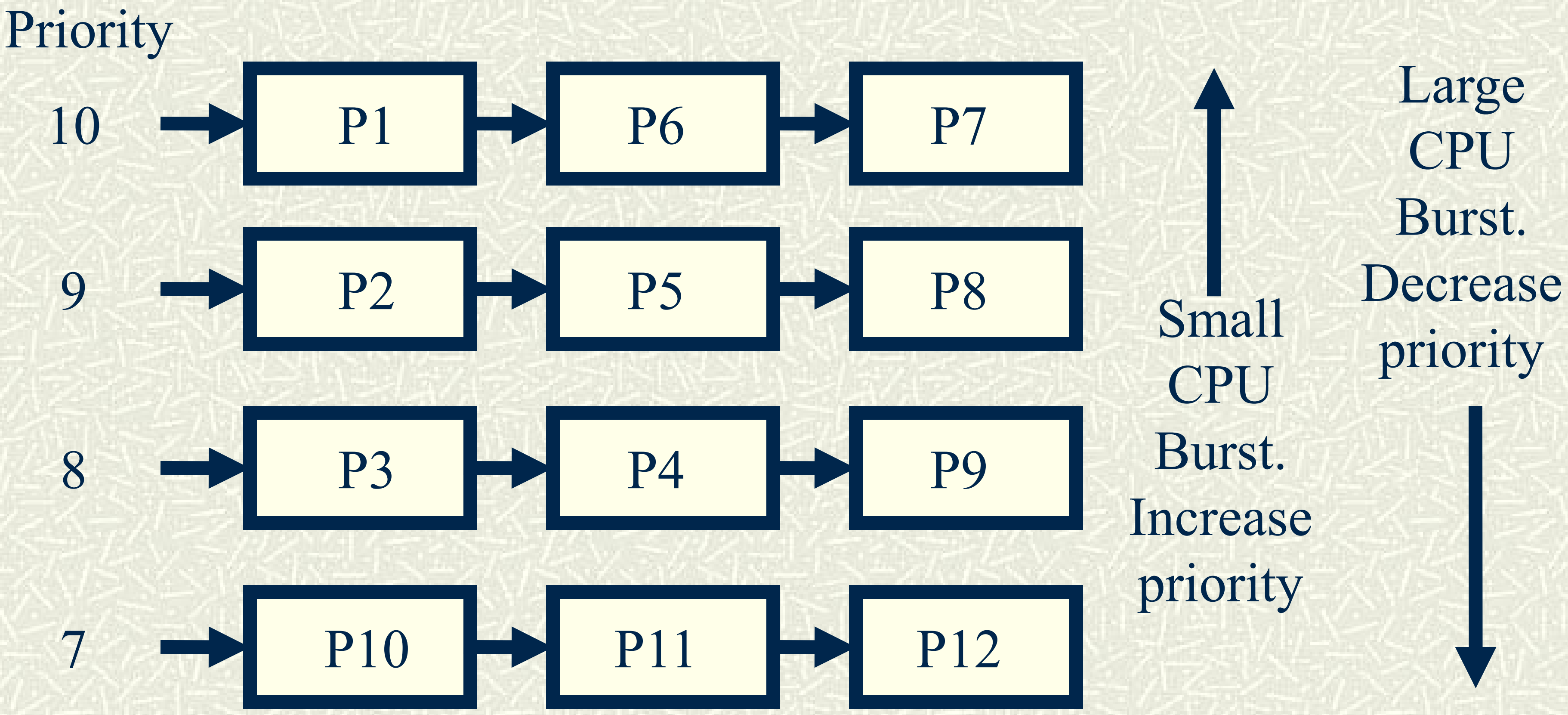
Scheduling Policies for <u>Preemptive Scheduling</u>

- **Multilevel Feedback-Queue Scheduling**
  - Instead of a having a single queue of ready processes, there are multiple queues of different priorities.
  - The scheduler will schedule the ready processes with the highest priority first.
  - Within processes of the same priority round robin is used.

Multilevel Feedback-Queue Scheduling

- ***Problem***: If you have processes of higher priority, low priority processes will never run.
- ***Solution***: ***Use Process Aging:*** The longer a process stays in a queue, the priority is artificially increased. The process will be scheduled at some point. After that, the priority is reset to the initial priority.
- Also, the scheduler benefits ***interactive processes*** by increasing their priority if they finish before the quantum expires.
- It also increase priority if the quantum expires before the burst finishes.
- ***The smaller the CPU burst estimate, the higher the priority. The larger the CPU burst estimate, the lower***

Multilevel Feedback-Queue Scheduling


Priority
10
P1
P6
P7
9
P2
P5
P8
8
P3
P4
P9
7
P10
P11
P12
Small CPU Burst. Increase priority
Large CPU Burst. Decrease priority

Threads and Thread Synchronization

Introduction to Threads

- A thread is a path execution
- By default, a C/C++ program has one thread called "main thread" that starts the main() function.

```
main()
    ---
    ---
    printf( "hello\n" );
    ---
}
```

Introduction to Threads

- You can create multiple paths of execution using:
  - POSIX threads (standard)

```
pthread_create( &thr_id, attr,
                func, arg )
```

  - Solaris threads

```
thr_create( stack, stack_size, func,
  arg, flags, &thr_id )
```

  - Windows

```
CreateThread(attr, stack_size, func,
    arg, flags, &thr_id)
```

Introduction to Threads

- Every thread will have its own
  - Stack
  - PC – Program counter
  - Set of registers
  - State
- Each thread will have its own function calls, and local variables.
- The process table entry will have a stack, set of registers, and PC for every thread in the process.

Multi-threaded Program Example

```
#include <pthread.h>
void prstr( char *s ){
  while( 1 ){
    printf( "%s",s);
   }
}
int main(){
  // thread 2
  pthread_create( NULL, NULL, prstr, "b\n" );
  // thread 3
  pthread_create(NULL, NULL, prstr, "c\n" );
  // thread 1
   prstr( "a\n" );
}
```

Multi-threaded Program Example

T2

```
void prstr( char *s ){
  while( 1 ){
    printf( "%s",s);
  }
}
```

T3

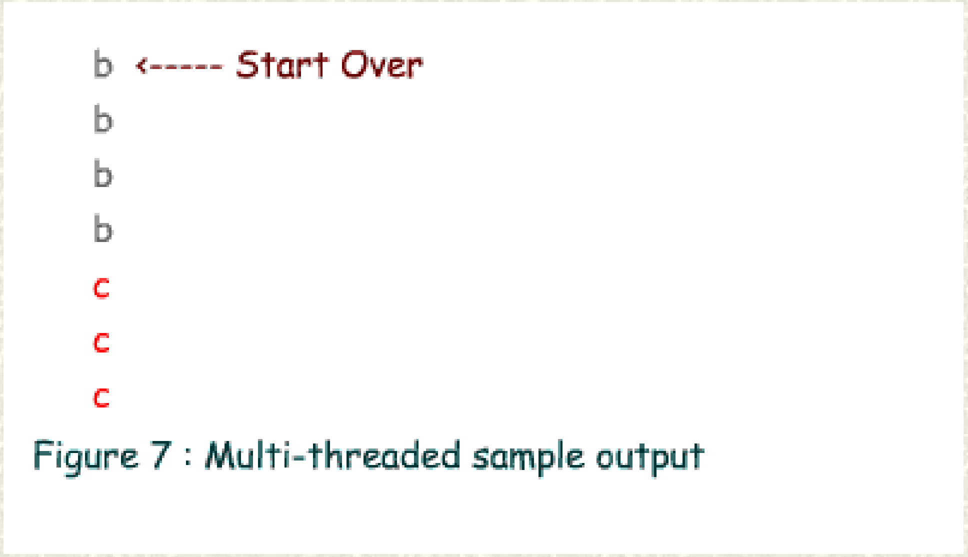
```
void prstr( char *s ){
  while( 1 ){
    printf( "%s",s);
  }
}
```

T1

```
main():
void prstr( char *s ){
  while( 1 ){
    printf( "%s",s);
  }
}
```

Multi-threaded Program Example

Output:



```
b <----- Start Over
b
b
b
c
c
c
```

Figure 7 : Multi-threaded sample output

Applications of Threads

- Concurrent Server applications
  - Assume a web server that receives two requests:
  - First, one request from a computer connected through a modem that will take 2 minutes.
  - Then another request from a computer connected to a fast network that will take .01 secs.
  - If the web server is single threaded, the second request will be processed only after 2 minutes.
  - In a multi-threaded server, two threads will be created to process both requests simultaneously. The second request will be processed as soon as it arrives.

Application of Threads

- Taking Advantage of Multiple CPUs
  - A program with only one thread can use only one CPU. If the computer has multiple cores, only one of them will be used.
  - If a program divides the work among multiple threads, the OS will schedule a different thread in each CPU.
  - This will make the program run faster.

Applications of Threads

- Interactive Applications.
  - Threads simplify the implementation of interactive applications that require multiple simultaneous activities.
  - Assume an Internet telephone application with the following threads:
    - Player thread - receives packets from the internet and plays them.
    - Capture Thread – captures sound and sends the voice packets
    - Ringer Server – Receives incoming requests and tells other phones when the phone is busy.
  - Having a single thread doing all this makes the code cumbersome and difficult to read.

Advantages and Disadvantages of Threads vs. Processes

- Advantages of Threads
  - ***Fast thread creation*** - creating a new path of execution is faster than creating a new process with a new virtual memory address space and open file table.
  - ***Fast context switch*** - context switching across threads is faster than across processes.
  - ***Fast communication across threads*** – threads communicate using global variables that is faster and easier than processes communicating through pipes or files.

Advantages and Disadvantages of Threads vs. Processes

- Disadvantages of Threads
  - ***Threads are less robust than processes*** – If one thread crashes due to a bug in the code, the entire application will go down. If an application is implemented with multiple processes, if one process goes down, the other ones remain running.
  - ***Threads have more synchronization problems*** – Since threads modify the same global variables at the same time, they may corrupt the data structures. Synchronization through mutex locks and semaphores is needed for that. Processes do not have that problem because each of them have their own copy of the variables.

Synchronization Problems with Multiple Threads

- Threads share same global variables.
- Multiple threads can modify the same data structures at the same time
- This can corrupt the data structures of the program.
- Even the most simple operations, like increasing a counter, may have problems when running multiple threads.

Example of Problems with Synchronization

```
// Global counter
int counter = 0;
void increment_loop(int max){
  for(int i=0;i<max;i++){
    int tmp = counter;
    tmp=tmp+1;
    counter=tmp;
  }
}
```

Example of Problems with Synchronization

```
int main(){
  pthread_t t1,t2;
  pthread_create(&t1,NULL,
                increment_loop,10000000);
  pthread_create(&t2,NULL,
                increment_loop,10000000);
  //wait until threads finish
  pthread_join(&t1);
  pthread_join(&t2);
  printf("counter total=%d",counter);
}
```

Example of Problems with Synchronization

- We would expect that the final value of counter would be `10,000,000+ 10,000,000= 20,000,000` ***but very likely the final value will be less than that (E.g. 19,998,045).***
- The context switch from one thread to another may change the sequence of events so the counter may loose some of the counts.

Example of Problems with Synchronization

T1

```
int counter = 0;
void increment_loop(int max){
  for(int i=0;i<max;i++){
   a)int tmp = counter;
    b)tmp=tmp+1;
    c)counter=tmp;
  }
}
```

T2

```
int counter = 0;
void increment_loop(int max){
  for(int i=0;i<max;i++){
    a)int tmp= counter;
    b)tmp=tmp+1;
    c)counter=tmp;
  }
}
```

Example of Problems with Synchronization

time

T1	T2	T0 (main)
for(…) a) tmp1=counter (tmp1=0) (Context switch)		Join t1 (wait)
	Starts running a) tmp2=counter (tmp2=0) b) tmp2=tmp2+1 c) counter=tmp2 Counter=1 a)b)c)a)b)c)… Counter=23 (context switch)	
b) tmp1=tmp1+1 c) counter=tmp1 Counter=1		

Example of Problems with Synchronization

- As a result 23 of the increments will be lost.
- T1 will reset the counter variable to 1 after T2 increased it 23 times.
- If this happens to a simple increment variables, worst things will happen to lists, hash tables and other data structures in a multi-threaded program.
- The solution is to make certain pieces of the code Atomic.
- Even if we use counter++ instead of a)b) c) we still have the same problem because the compiler will generate separate instructions that will look like a)b)c).

Atomicity

Atomic Section:

- ***A portion of the code that only one thread should execute at a time while the other threads have to wait.***
- ***Otherwise corruption of the variables is possible.***
- ***An atomic section is also called sometimes a "Critical Section"***

Mutex Locks

- ***Mutex Locks*** are software mechanisms that enforce atomicity
- Declaration:

```
#include <pthread.h>
pthread_mutex_t mutex;
```

- Initialize

```
pthread_mutex_init( &mutex, atttributes);
```

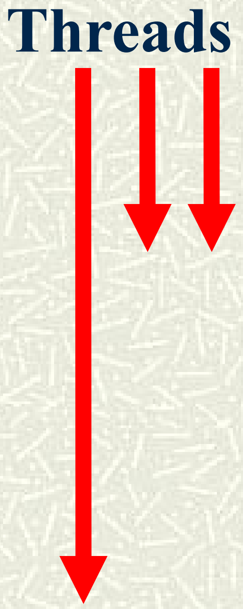
- Start Atomic Section

```
pthread_mutex_lock(&mutex);
```

- End Atomic section

```
pthread_mutex_unlock(&mutex);
```

Example of Mutex Locks



```
#include <pthread.h>
int counter = 0; // Global counter
pthread_mutex_t mutex;
void increment_loop(int max){
  for(int i=0;i<max;i++){
    pthread_mutex_lock(&mutex);
    int tmp = counter;
    tmp=tmp+1;
    counter=tmp;
    pthread_mutex_unlock(&mutex);
  }
}
```

Example of Mutex Locks

```
int main(){
  pthread_t t1,t2;
  pthread_mutex_init(&mutex,NULL);
  pthread_create(&t1,NULL,
                 increment,10000000);
  pthread_create(&t2,NULL,
                 increment,10000000);
  //wait until threads finish
  pthread_join(&t1);
  pthread_join(&t2);
  printf("counter total=%d",counter);
}
```

Example of Mutex Locks

time

T1	T2	T0 (main)
for(…) mutex_lock(&m) a) tmp1=counter (tmp1=0) (Context switch)		Join t1 (wait)
	Starts running mutex_lock(&m) (wait) (context switch)	

b) tmp1=tmp1+1
c) counter=tmp1
Counter=1
mutex_unlock(&m)		

	a) tmp2=counter b) tmp2=tmp2+1 c) counter=tmp2	

Example of Mutex Locks

- As a result, the steps a),b),c) will be atomic so the final counter total will be `10,000,000+ 10,000,000= 20,000,000` no matter if there are context switches in the middle of a)b)c)

Mutual Exclusion

- Mutex Locks also enforce the mutual exclusion of two or more sections of code.

```
Mutex_lock(&m)            Mutex_lock(&m)
A                         D
B                         E
C                         F
Mutex_unlock(&m)          Mutex_unlock(&m)
```

Mutual Exclusion

- This means that the sequence ABC, DEF, can be executed as an atomic block without interleaving.

```
Time ---------------------------->
T1 -> ABC              ABC
T2 ->       DEF           DEF
T3 ->          ABC DEF
```

Mutual Exclusion

- If different mutex locks are used then the sections are still atomic but they are not mutually exclusive (m1!=m2)

```
Mutex_lock(&m1)
A
B
C
Mutex_unlock(&m1)
```

```
Mutex_lock(&m2)
D
E
F
Mutex_unlock(&m2)
```

Mutual Exclusion

- This means that the sequence ABC, and DEF are atomic but they can interleave each other,

```
Time ------------------------->

T1 -> AB   C            A   BC

T2 ->     D   EF          D  EF

T3 ->           ABC           DEF
```

Mutex Lock implementation

- Two approaches:
  - Disable Interrupts
    - Do not allow context switches to happen.
    - Available only in Kernel mode
    - Other interrupts may be lost.
    - Only used in kernel programming
    - Slow if used in a machine with more than 1 processor
  - Spin Locks
    - Spin locks use an instruction test_and_set assembly instruction that is guaranteed to be atomic.
    - Most architectures provide this instruction.
    - Spin locks can be implemented in user space

Spin Locks

- There is an instruction test_and_set that is guaranteed to be atomic
- Pseudocode:

```
int test_and_set(int *v){
   int oldval = *v;
   *v = 1;
   return oldval;
}
```

- This instruction is implemented by the CPU. You don’t need to implement it.

Spin Locks

- Mutex locks are implemented on top of spinlocks.
- Spinlocks make thread “spin” busy waiting until lock is released, instead of putting thread in waiting state.

Spin Locks

```
int lock = 0;
void spinlock(int * lock)
{
  while (test_and_set(&lock) != 0) {
    // Give up CPU
     thr_yield();
  }
}
void spinunlock(int*lock){
  *lock = 0;
}
```

Example of Spin Locks

```
#include <pthread.h>
int counter = 0; // Global counter
int m = 0;
void increment_loop(int max){
  for(int i=0;i<max;i++){
    spin_lock(&m);
    a) int tmp = counter;
    b) tmp=tmp+1;
    c) counter=tmp;
    spin_unlock(&m);
  }
}
```

Spin Locks Example

T1	T2	T0

for(…)
spin_lock(&m)
while (test_and_set(&m))
→ oldval=0 (m=1)break while
a)
(Context switch)
		Join t1
		Join t2
		(wait)

	Starts running
	spin_lock(&m)
	while (test_and_set(&m))
	->oldval=1 (m==1)continue in while
	thr_yield()(context switch)	

b)c) Counter=1 spin_unlock(&m) m=0		
	while (test_and_set(&m)) -> oldval=0 Break while a) b) c)	

Spin Locks

- You could use spinlocks as a mutex locks.
- However a spinlock has the disadvantage of wasting CPU cycles doing busy waiting.
- Mutex locks use Spin Locks but they put threads in wait state instead of spinning.
- The overhead in spinlocks decreases by calling pthread_yield to give up the CPU. But not entirely.
- Spin Locks can be used as mutex locks as long as the critical sections are small and take very short time.
- Spinlocks are not fair but mutexes are fair.

Mutex Lock and Spin Locks

```
mutex_lock(mutex) {
  spin_lock();
  if (mutex.lock) {
    mutex.queue(
      currentThread)
    spin_unlock();
    currentHread.
         setWaitState();
    GiveUpCPU();
  }
  mutex.lock = true;
  spin_unlock();
}
```

```
mutex_unlock() {
  spin_lock();
  if (mutex.queue.
       nonEmpty) {
    t=mutex.dequeue();
    t.setReadyState();
  }
  else {
    mutex.lock=false;
  }
  spin_unlock();
}
```

Threads and Fork

- The behavior of fork() changes depending on the flavor of threads you use.
- When fork() is called, threads created with pthread_create() are not duplicated except if it is the thread calling fork().
- In Solaris threads created with thr_create() are duplicated in child too.

Threads and Fork

- Example:
  - In Solaris one process creates 5 threads using thr_create and then calls fork() then the child will also have a copy of the 5 threads so we have a total of 10 threads
  - In Solaris (or other UNIX flavors) one process creates 5 threads using pthread create and then calls fork() therefore the child will only get one thread that is the thread that called fork so we have a total of 6 threads
- Solaris has a system call fork1() that only copies the calling thread in the child process.
- Modern UNIX use the pthread_create semantics that only copies the calling thread.

Thread Safe and Race Condition

- Data structures and functions that are able to handle multiple threads are called ***"Thread Safe" or "Thread Asynch" or "Reentrant" or "Multi-Threaded (MT).***
- A ***Thread Unsafe*** function or data structure is one that can only be used by a single thread.
- A bug related to having multiple threads modifying a data structure simultaneously is called a ***"Race Condition"***.
- ***Race Conditions*** are difficult to debug because they are often difficult to reproduce.

A Thread Unsafe List Class

```
#include <pthread.h>

struct ListEntry {
   int _val;
   ListEntry *_next;
};
class List{
  ListEntry *_head;
public:
   List();
   void insert(int val);
   int remove();
}
```

```
List::List(){
  _head = NULL;
}

List::insert(int val){
  ListEntry *e =
        new ListEntry;

   e->_val = val;

  a) e->_next = _head;
  b)_head = e;

}
```

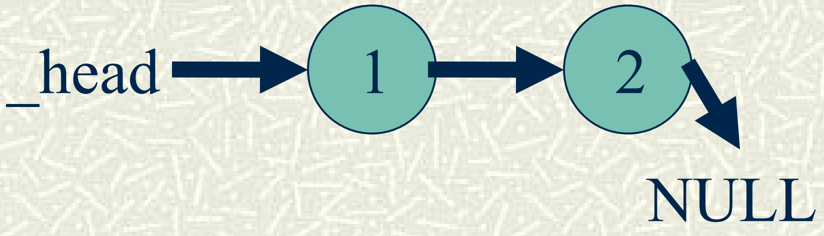
A Thread Unsafe List Class

```
int List::remove(){
  ListEntry *tmp;

   c) tmp = _head;

   if(tmp == NULL) {
     return -1;
   }

   d) _head=tmp->_next;

  int val=tmp->_val;
  delete tmp;
  return val;
}
```

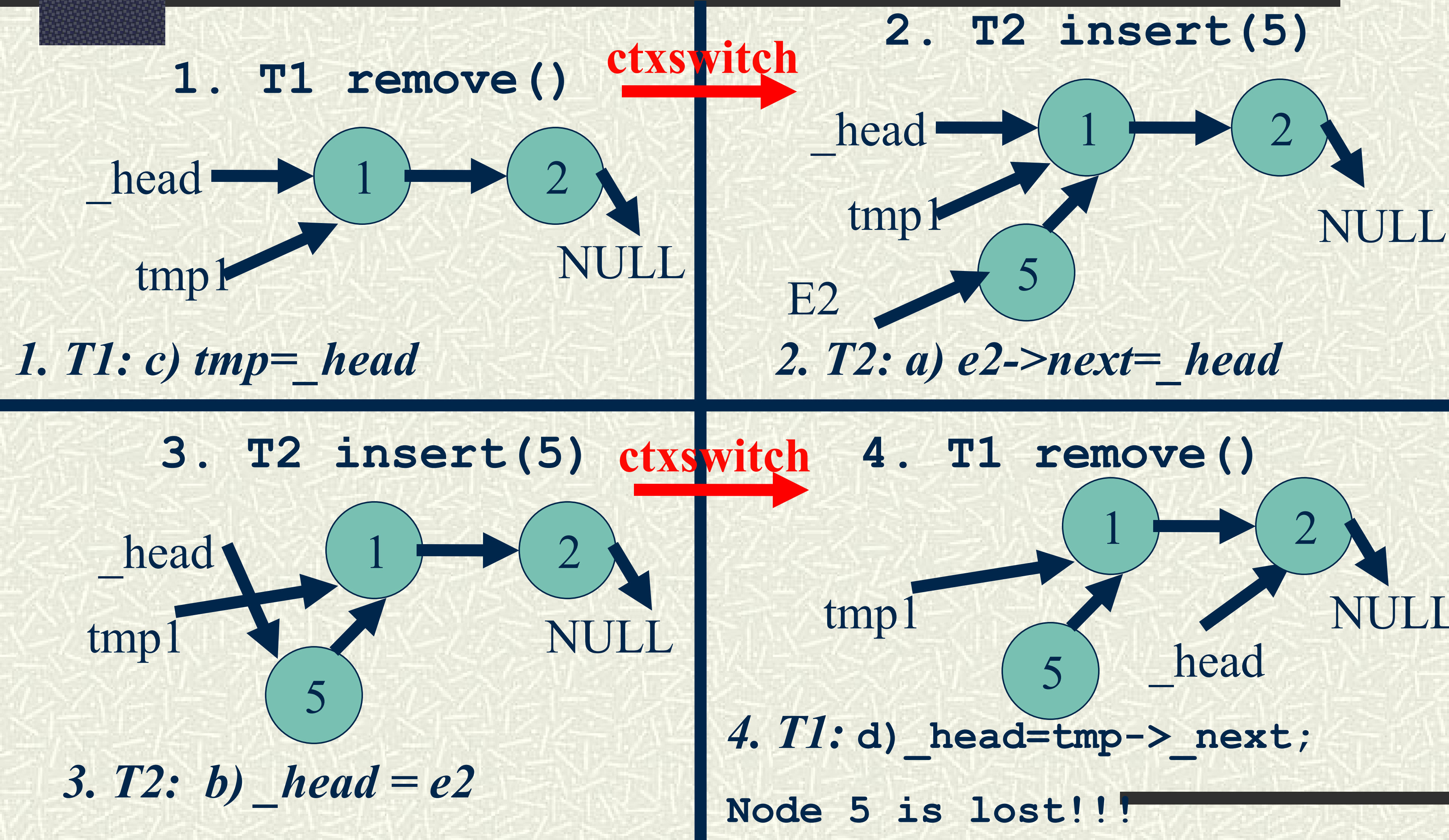
Race Conditions in List

- Assume our list did not have mutex locks.
- We can have the following race condition: Assume T1 calls remove() and T2 calls insert().

Initially



Race Conditions in List



An Thread Safe List Class

```
#include <pthread.h>

struct ListEntry {
  int _val;
  ListEntry *_next;
};
class MTList{
  ListEntry *_head;
  pthread_mutex_t _mutex;
public:
  MTList();
  void insert(int val);
  int remove();
}
```

```
MTList::MTList(){
  _head = NULL;
  pthread_mutex_init(
     &_mutex, NULL );
}

MTList::insert(int val){
   ListEntry *e =
         new ListEntry;
   e->_val = val;
   pthread_mutex_lock(&_mutex);
  a) e->_next = _head;
  b)_head = e;
  pthread_mutex_unlock(&mutex);
}
```

A Thread Safe List

```
int MTList::remove(){
  ListEntry *tmp;
  pthread_mutex_lock(&_mutex);
  c) tmp = _head;
  if(tmp == NULL) {
    pthread_mutex_unlock(&_mutex);
    return -1;
  }
  d) _head=tmp->_next;
  pthread_mutex_unlock(&mutex);
  int val=tmp->_val;
  delete tmp;
  return val;
}
```

Drawbacks in the List Class

- The MT List implementation described before does not behave well if the list is empty (returning –1).
- We would like an implementation where remove waits if the List is empty.
- This behavior cannot be implemented with only mutex locks
- Semaphores are needed to implement this behavior.

Semaphores

- Semaphores are a generalization of mutexes.
- A mutex allows only one thread to execute the code guarded by a mutex_lock / mutex_unlock pair.
- A semaphore allows a maximum number of threads between a sema_wait, sema_post.
- A semaphore is initialized with an initial counter that is greater or equal to 0.
- You could implement a mutex using a semaphore with an initial counter of 1.

Semaphore Calls

- Declaration:

  #include <synch.h>

  sema_t sem;

- Initialization:

  sema_init(sema_t *sem, int counter, int type, void
      * arg));

- Decrement Semaphore:

  sema_wait(sema_t *sem);

- Increment Semaphore

  sema_post(sema_t *sem);

- Man pages

  man sema_wait

Semaphore Calls Implementation

```
Pseudocode:
sema_init(sem_t *sem, counter){
  sem -> count = counter;
}

sema_wait(sema_t *sem){
  sem ->count--;
  if(sem ->count < 0){
     wait();
  }
}
```

Semaphore Calls Implementation

```
sema_post(sema_t *sem){
  sem ->count++;
  if(there is a thread waiting){
     wake up the thread that
     has been waiting the longest.
     }
}
```

Note: The mutex lock calls are omitted in the pseudocode for simplicity.

Use of Semaphore Counter

- Mutex Lock Case: initial counter == 1
  - Equivalent to a Mutex Lock
- N Resources Case: initial counter is n > 1
  - Control access to a fixed number n of resources. Example: Access to 5 printers. 5 computers can use them. 6th computer will need to wait.
- Wait for an Event Case: initial counter == 0
  - Synchronization. Wait for an event. A thread calling sema_wait will block until another threads sends an event by calling sema_post.

Example Semaphore count=1 (Mutex Lock Case)

```
Assume sema_t sem; sema_init(&sem, 1);
T1                          T2
sema_wait(&sem)
 sem->count--(==0)
 Does not wait
 ..
 ctxswitch                  sema_wait(&sem)
                            sem->count--(==-1)
                            if (sem->count < 0)
                              wait (ctxswitch)
 ..
sema_post(&sem)
 sem->count++(==0)
 wakeup T2                   continue
```

Example Semaphore count=3 (N Resources Case)

```
Assume sema_t sem; sema_init(&sem, 3); (3 printers)
T1                  T2                T3
sema_wait(&sem)
 sem->count--(==2)
 Does not wait
 print
 ..                  sema_wait(&sem)
                     sem->count--(==1)
                     Does not wait
                     print
                     ..
                                      sema_wait(&sem)
                                      sem->count--(==0)
                                      Does not wait
                                      print
```

Example Semaphore count=3 (N Resources Case)

```
T4                          T5                        T1

  sema_wait(&sem)
   sem->count--(==-1)
   wait
                          sema_wait(&sem)
                          sem->count--(==-2)
                          Wait
                                                   Finished printing
                                                   sema_post(&sem)
                                                   sem->count++(==-1)
                                                   Wakeup T4
   print
```

Example Semaphore count=0 Wait for an Event Case

```
Assume sema_t sem; sema_init(&sem, 0);
T1                          T2
// wait for event
sema_wait(&sem)
 sem->count--(==-1)
  wait

                            //Send event to t1
                            sema_post(&sem)
                            sem->count++(==0)
                            Wakeup T1
                            ..
T1 continues
```

A Synchronized List Class

- We want to implement a List class where remove() will block if the list is empty.
- To implement this class, we will use a semaphore "_emptySem" that will be initialized with a counter of 0.
- Remove() will call sema_wait(&_emptySem) and it will block until insert() calls sema_post(&emptySem).
- The counter in semaphore will be equivalent to the number of items in the list.

A Synchronized List Class

```
SyncList.h
#include <pthread.h>
struct ListEntry {
   int _val;
    ListEntry *_next;
};
class SyncList{
   ListEntry *_head;
   pthread_mutex_t _mutex;
   sema_t _emptySem;
public:
    SyncList();
    void insert(int val);
    int remove();
};
```

```
SynchList.cc
SyncList:: SyncList(){ _
   _head = NULL;
   pthread_mutex_init(
      &_mutex, NULL );
   sema_init(&_emptySem,0,
     USYNC_THREAD, NULL);
}
SyncList ::insert(int val){
   ListEntry *e =
          new ListEntry;
    e->_val = val;
    pthread_mutex_lock(&_mutex);
   a)e->_next = _head;
   b)_head = e;
   pthread_mutex_unlock(&_mutex);
   sema_post(&_emptySem);
```

A Synchronized List

```
int SyncList ::remove(){
  ListEntry *tmp;
  // Wait until list is not empty
  sema_wait(&_emptySem);
  pthread_mutex_lock(&_mutex);
   c)tmp = _head;
   d)head=tmp->_next;
   pthread_mutex_unlock(&mutex);
  int val=tmp->_val;
  delete tmp;
  return val;
}
```

Example: Removing Before Inserting

```
T1                            T2
remove()
sema_wait(&_emptySem)
(count==-1)
  wait

                          insert(5)
                          pthread_mutex_lock()
                          a) b)
                          pthread_mutex_unlock()
                          sema_post(&_emptySem)
                   <----- wakeup T1
T1 continues
pthread_mutex_lock()
c) d)
pthread_mutex_unlock()
```

Example: Inserting Before Removing

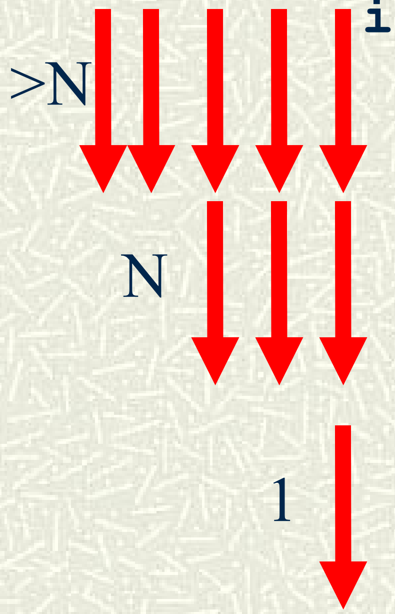
T1 | **T2**

```
                                          insert(7)
                                          pthread_mutex_lock()
                                          a) b)
                                          pthread_mutex_unlock()
                                          sema_post(&_emptySem)(count==1)

Starts running
remove()
sema_wait(_emptySem);
continue (count==0)
pthread_mutex_lock()
c) d)
pthread_mutex_unlock()
```

Notes on Synchronized List Class

- We need the mutex lock to enforce atomicity in the critical sections a) b) and c) d).
- The semaphore is not enough to enforce atomicity.

Notes on Synchronized List Class



```
int MTList::remove(){
  ListEntry *tmp;
  sema_wait(&_emptySem);

  pthread_mutex_lock(&_mutex);
  c)tmp = _head;
  d)head=tmp->_next;
  pthread_mutex_unlock(&mutex);
  int val=tmp->_val;
  delete tmp;
  return val;
}
```

(N= items in list)

Notes on Synchronized List Class

- Te sema_wait() call has to be done outside the mutex_lock/unlock section.
- Otherwise we can get a deadlock.
- Example:

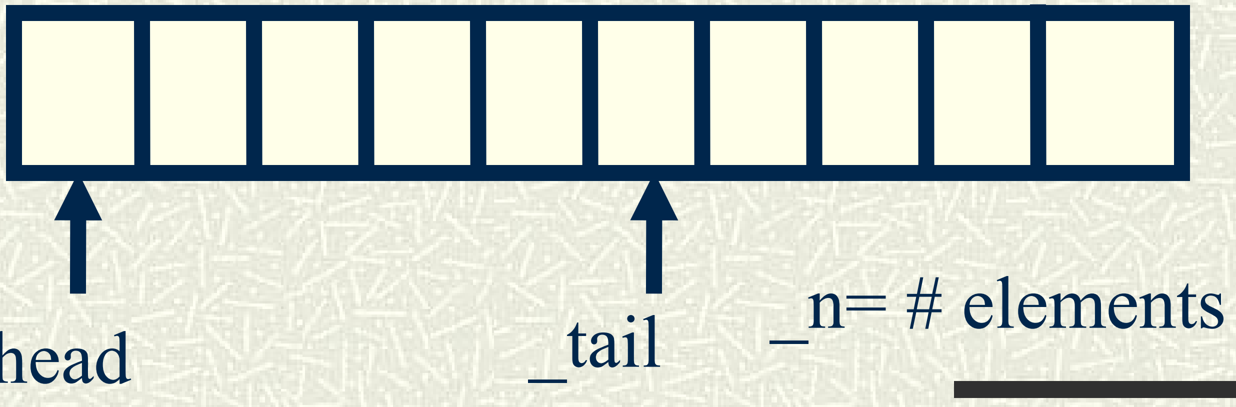
```
pthread_mutex_lock(&_mutex);
sema_wait(&_empty);
  c)tmp = _head;
  d)head=tmp->_next;
  pthread_mutex_unlock(&mutex);
```

Notes on Synchronized List Class

```
     T1                               T2
    remove()
    mutex_lock
    sema_wait(&_empty)
      wait for sema_post
      from T2

                                 insert(5)
                                 pthread_mutex_lock()
                                 wait for mutex_unlock in T1

                     Deadlock!!!!
```

Bounded Buffer

- Assume we have a circular buffer of a maximum fixed size.
- We want multiple threads to communicate using this circular buffer.
- It is commonly used in device drivers and in pipes.



Bounded Buffer

- The queue has two functions
  - enqueue() - adds one item into the queue. It blocks if queue if full
  - dequeue() - remove one item from the queue. It blocks if queue is empty
- Strategy:
  - Use an _emptySem semaphore that dequeue() will use to wait until there are items in the queue
  - Use a _fullSem semaphore that enqueue() will use to wait until there is space in the queue.

Bounded Buffer

```
#include <pthread.h>
enum {MaxSize = 10};
class BoundedBuffer{
  int _queue[MaxSize];
  int _head;
  int _tail;
  mutex_t _mutex;
  sema_t _emptySem;
  sema_t _fullSem;
public:
  BoundedBuffer();
  void enqueue(int val);
  int dequeue();
};
```

```
BoundedBuffer::BoundedBuffer() {
  _head = 0;
  _tail = 0;
  pthread_mutex_init(&_mutex,
   NULL, NULL);
  sema_init(&_emptySem, 0
    USYNC_THREAD, NULL);
  sema_init(&_fullSem, MaxSize
    USYNC_THREAD, NULL);
}
```

Bounded Buffer

```
void
BoundedBuffer::enqueue(int val)
{
  sema_wait(&_fullSem);
  mutex_lock(_mutex);
  _queue[_tail]=val;
  _tail = (_tail+1)%MaxSize;
  mutex_unlock(_mutex);
  sema_post(_emptySem);
}
```

```
int
BoundedBuffer::dequeue()
{
  sema_wait(&_emptySem);
  mutex_lock(_mutex);
  int val = _queue[_head];
  _head = (_head+1)%MaxSize;
  mutex_unlock(_mutex);
  sema_post(_fullSem);
  return val;
}
```

Bounded Buffer

```
Assume queue is empty
T1                   T2                        T3
v=dequeue()
sema_wait(&_emptySem);
 _emptySem.count==-1
 wait
                  v=dequeue()
                  sema_wait(&_emptySem);
                  _emptySem.count==-2
                  wait
                                      enqueue(6)
                                      sema_wait(&_fullSem)
                                      put item in queue
                                      sema_post(&emptySem)
                                      _emptySem.count==-1
                                      wakeup T1
T1 continues
Get item from queue
```

Bounded Buffer

```
Assume queue is empty

T1                    T2           ……               T10

enqueue(1)
sema_wait(&_fullSem);
 _fullSem.count==9
 put item in queue
                  enqueue(2)
                  sema_wait(&_fullSem);
                  _fullSem.count==8
                  put item in queue
                                        enqueue(10)
                                        sema_wait(&_fullSem);
                                        _fullSem.count==0
                                        put item in queue
```

Bounded Buffer

```
T11                     T12
enqueue(11)
sema_wait(&_fullSem);
 _fullSem.count==-1
 wait
                     val=dequeue()
                     sema_wait(&_emptySem);
                     _emptySem.count==9
                     get item from queue
                     sema_post(&_fullSem)
                     _fullSem.count==0
                     wakeup T11
```

Bounded Buffer Notes

- The counter for _emptySem represents the number of items in the queue
- The counter for _fullSem represents the number of spaces in the queue.
- Mutex locks are necessary since sema_wait(_emptySem) or sema_wait(_fullSem) may allow more than one thread to execute the critical section.

Read/Write Locks

- They are locks for data structures that can be read by multiple threads simultaneously (multiple readers) but that can be modified by only one thread at a time.
- Example uses: Data Bases, lookup tables, dictionaries etc where lookups are more frequent than modifications.

Read/Write Locks

- **Multiple readers** may read the data structure simultaneously
- **Only one writer** may modify it and it needs to exclude the readers.
- Interface:

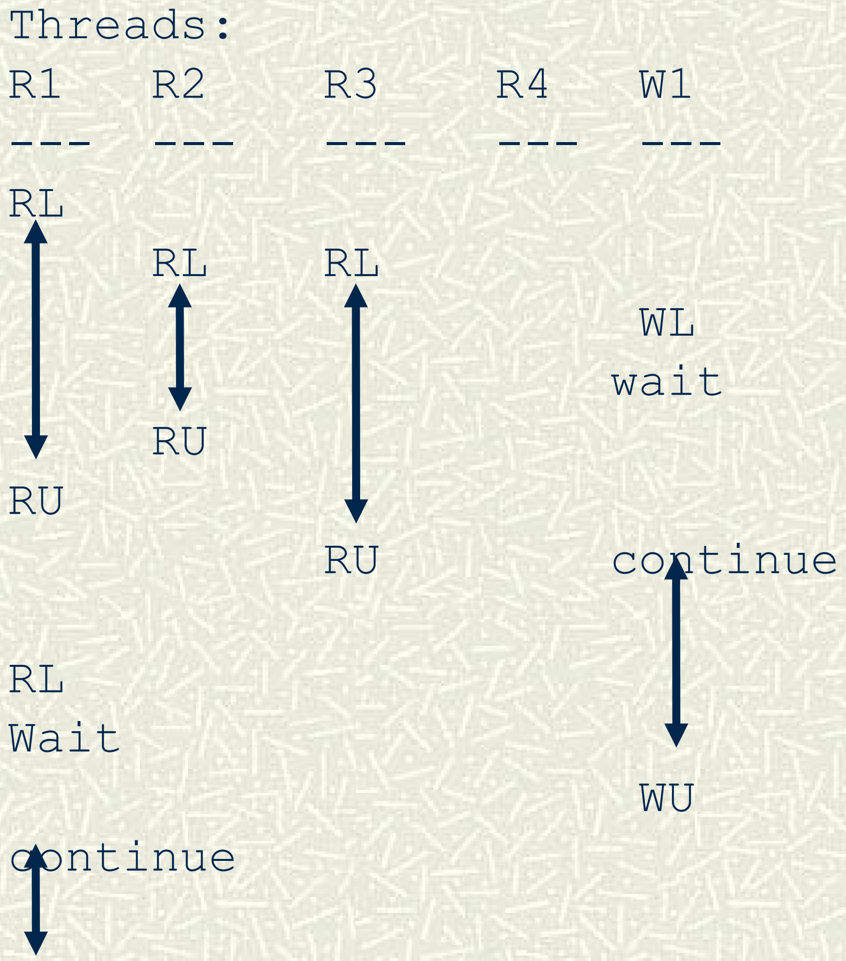
ReadLock() – Lock for reading. Wait if there are writers holding the lock

ReadUnlock() – Unlock for reading

WriteLock() - Lock for writing. Wait if there are readers or writers holding the lock

WriteUnlock() – Unlock for writing

Read/Write Locks



```
rl = readLock;
ru = readUnlock;
wl = writeLock;
wu = writeUnlock;
```

Read/Write Locks Implementation

```
class RWLock
{
   int _nreaders;
   //Controls access
   //to readers/writers
   sema_t _semAccess;
   mutex_t _mutex;
public:
   RWLock();
   void readLock();
   void writeLock();
   void readUnlock();
   void writeUnlock();
};
```

```
RWLock::RWLock()
{
   _nreaders = 0;
   sema_init( &semAccess, 1 );
   mutex_init( &_mutex );
}
```

Read/Write Locks Implementation

```
void RWLock::readLock()
{
   mutex_Lock( &_mutex );
   _nreaders++;
   if( _nreaders == 1 )
   {
       //This is the
       // first reader
       //Get sem_Access
       sem_wait(&_semAccess);
   }
   mutex_unlock( &_mutex );
}
```

```
void RWLock::readUnlock()
{
   mutex_lock( &_mutex );
   _nreaders--;
   if( _nreaders == 0 )
   {
       //This is the last reader
       //Allow one writer to
       //proceed if any
       sema_post( &_semAccess );
   }
   mutex_unlock( &_mutex );
}
```

Read/Write Locks Implementation

```
void RWLock::writeLock()
{
   sema_wait( &_semAccess );
}
```

```
void RWLock::writeUnlock()
{
   sema_post( &_semAccess );
}
```

Read/Write Locks Example

```
Threads:
R1            R2              R3             W1         W2
-----------   ------------  --------      --------  --------
readLock
  nreaders++(1)
  if (nreaders==1)
    sema_wait
    continue
             readLock
               nreaders++(2)
                          readLock
                            nreaders++(3)
                                        writeLock
                                        sema_wait
                                        (block)
```

Read/Write Locks Example

```
Threads:
R1            R2            R3          W1       W2
-----------   ------------  --------   -------- --------
                                                 writeLock
                                                 sema_wait
                                                 (block)
readUnlock()
  nreaders—(2)
              readUnlock()
              nreaders—(1)
                            readUnlock()
                            nreaders—(0)
                            if (nreaders==0)
                              sema_post
                                      W1 continues

                                      writeUnlock
                                      sema_post
                                                 W2 continues
```

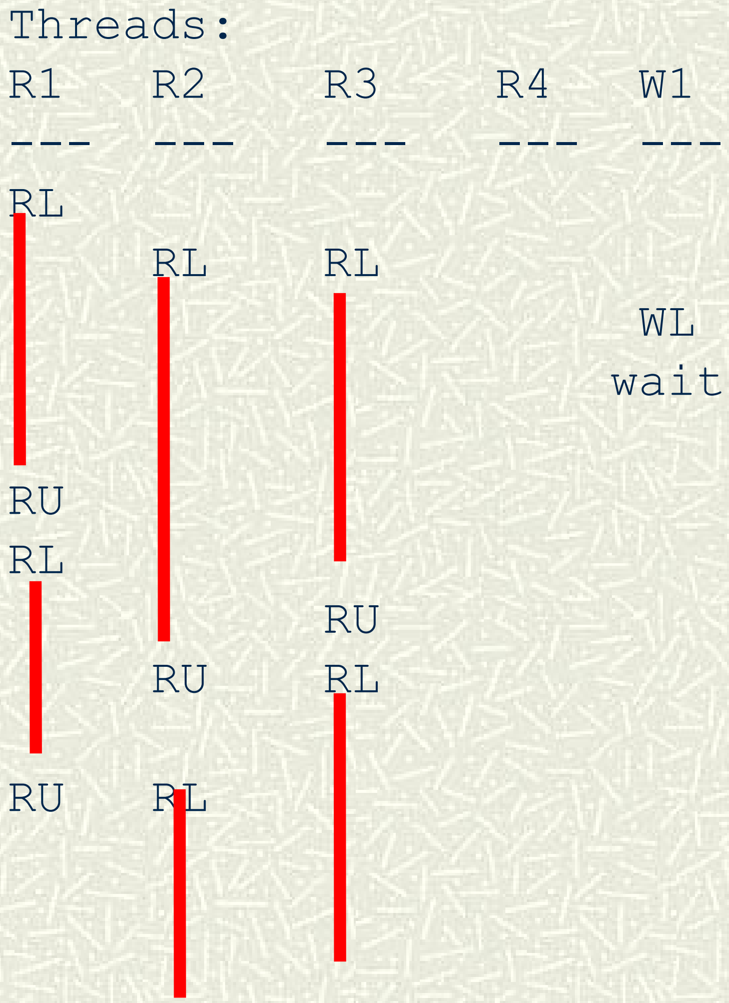
Read/Write Locks Example

```
Threads: (W2 is holding lock in write mode)
R1            R2            R3         W1        W2
-----------   ------------  --------   --------  --------
readLock
  mutex_lock
  nreaders++(1)
  if (nreaders==1)
    sema_wait
    block
             readLock
               mutex_lock
               block
                                                  writeUnlock
                                                    sema_post
R1 continues
  mutex_unlock
               R2 continues
```

Notes on Read/Write Locks

- Mutexes and semaphores are fair. The thread that has been waiting the longest is the first one to wake up.
- Spin locks are not fair, the first one that gets it will lock the spin lock.
- This implementation of read/write locks suffers from “starvation” of writers. That is, a writer may never be able to write if the number of readers is always greater than 0.

Write Lock Starvation (Overlapping readers)



```
Threads:
R1     R2     R3     R4     W1
---    ---    ---    ---    ---
RL
       RL     RL
                            WL
                           wait


RU
RL
              RU
       RU     RL

RU     RL
```

rl = readLock;
ru = readUnlock;
wl = writeLock;
wu = writeUnlock;

Deadlock and Starvation

- Deadlock
  - It happens when one or more threads will have to block forever (or until process is terminated) because they have to wait for a resource that will never be available.
  - Once a deadlock happens, the process has to be killed. Therefore we have to prevent deadlocks in the first place.
- Starvation
  - This condition is not as serious as a deadlock. Starvation happens when a thread may need to wait for a long time before a resource becomes available.
  - Example: Read/Write Locks

Example of a Deadlock

- Assume two bank accounts protected with two mutexes

```
int balance1 = 100;
int balance2 = 20;
mutex_t m1, m2;
```

```
Transfer1_to_2(int amount) {
  mutex_lock(&m1);
  mutex_lock(&m2);
  balance1 - = amount;
  balance2 += amount;
  mutex_unlock(&m1);
  mutex_unlock(&m2);
}
```

```
Transfer2_to_1(int amount) {
  mutex_lock(&m2);
  mutex_lock(&m1);
  balance2 - = amount;
  balance1 += amount;
  mutex_unlock(&m2);
  mutex_unlock(&m1);
}
```

Example of a Deadlock

```
Thread 1                                      Thread 2
-----------------------------------------     ---------------------------------------
Transfer1_to_2(int amount) {
  mutex_lock(&m1);
  context switch

                                              Transfer2_to_1(int amount) {
                                              mutex_lock(&m2);
                                              mutex_lock(&m1);
                                                 block waiting for m1



  mutex_lock(&m2);
  block waiting for m2
```

Example of a Deadlock

- Once a deadlock happens, the process becomes unresponsive. You have to kill it.
- Before killing get as much info as possible since this event usually is difficult to reproduce.
- the process attach the debugger to the processes to see where the deadlock happens.

```
 gdb progname <pid>
gdb> threads              //Lists all threads
gdb> thread <thread number>  //Switch to a thread
 gdb >where               // Prints stack trace
```

Do this for every thread.

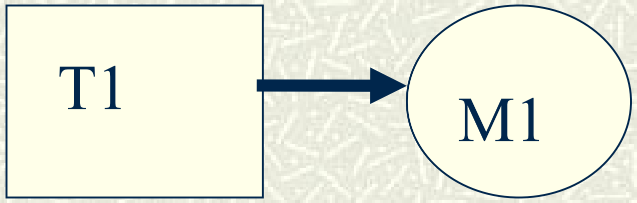
Then you can kill the process.

Deadlock

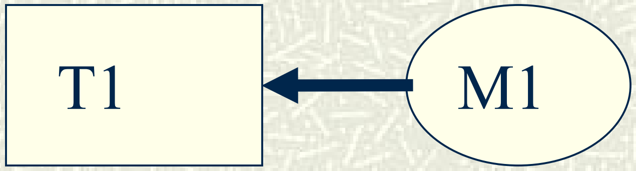
- A deadlock happens when there is a combination of instructions in time that causes resources and threads to wait for each other.
- You may need to run your program for a long time and stress them in order to find possible deadlocks
- Also you can increase the probability of a deadlock by running your program in a multi-processor (multi-core) machine.
- We need to prevent deadlocks to happen in the first place.

Graph Representation of Deadlocks

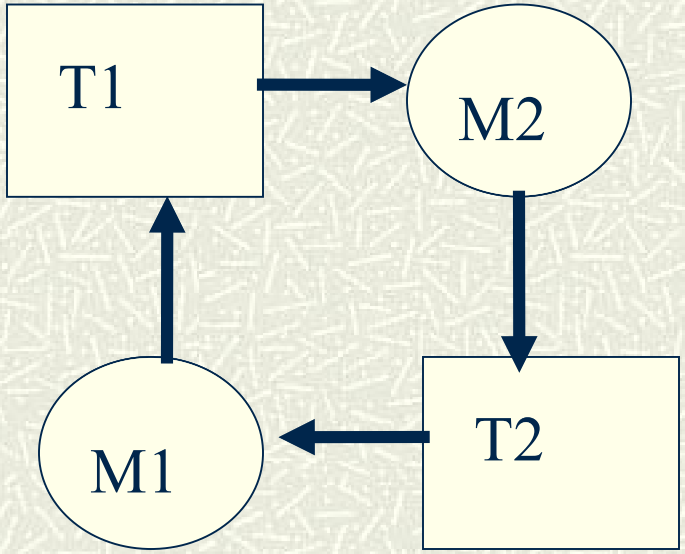
- Thread T1 is waiting for mutex M1



- Thread T1 is holding mutex M1

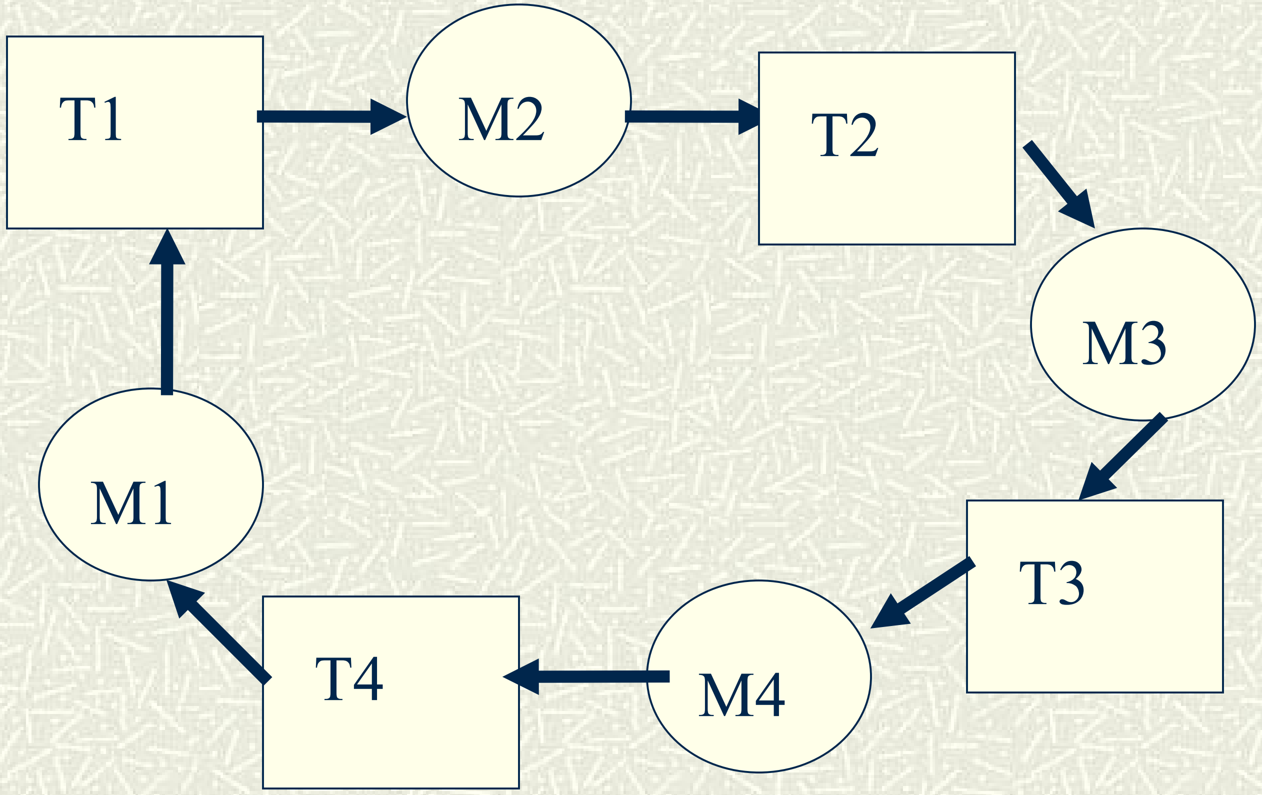


Deadlock Representation



Deadlock = Cycle in the graph.

Larger Deadlock



Deadlock Prevention

- A deadlock is represented as a cycle in the graph.
- ***To prevent deadlocks we need to assign an order to the locks: m1, m2, m3 …***
- Notice in the previous graph that a cycle follows the ordering of the mutexes except at one point.

Deadlock Prevention

- ***Deadlock Prevention:***

  When calling mutex_lock mi, lock the mutexes with lower order of I before the ones with higher order.
- If m1 and m3 have to be locked, lock m1 before locking m3.
- This will prevent deadlocks because this will force not to lock a higher mutex before a lower mutex breaking the cycle.

Lock Ordering => Deadlock Prevention

- Claim:
  - By following the lock ordering deadlocks are prevented.
- Proof by contradiction
  - Assume that the ordering was followed but we have a cycle.
  - By following the cycle in the directed graph we will find m_i before m_j. Most of the time $i< j$ but due to the nature of the cycle, at some point we will find $i > j$.
  - This means that a tread locked m_i before m_j where $i>j$ so it did not follow the ordering. This is a contradiction to our assumptions.
  - Therefore, lock ordering prevents deadlock.

Preventing a Deadlock

- Rearranging the Bank code to prevent the deadlock, we make sure that the mutex_locks are locked in order.

```
int balance1 = 100;
int balance2 = 20;
mutex_t m1, m2;

Transfer1_to_2(int amount) {
 mutex_lock(&m1);
 mutex_lock(&m2);
 balance1 -= amount;
 balance2 += amount;
 mutex_unlock(&m1);
 mutex_unlock(&m2);
}
```

```
Transfer2_to_1(int amount) {
 mutex_lock(&m1);
 mutex_lock(&m2);
 balance2 -= amount;
 balance1 += amount;
 mutex_unlock(&m2);
 mutex_unlock(&m1);
}
```

Preventing a Deadlock

- We can rewrite the Transfer function s more generically as:

```
int balance[MAXACOUNTS];
mutex_t mutex[MAXACOUNTS];

Transfer_i_to_j(int i, int
   j, int amount) {
  if ( i< j) {
     mutex_lock(&mutex[i]);
     mutex_lock(&mutex[j]);
  }
  else {
     mutex_lock(&mutex[j]);
     mutex_lock(&mutex[i]);
  }
balance1 -= amount;
  balance2 += amount;

   mutex_unlock(&mutex[i]
   );

   mutex_unlock(&mutex[j]
   );
}
```

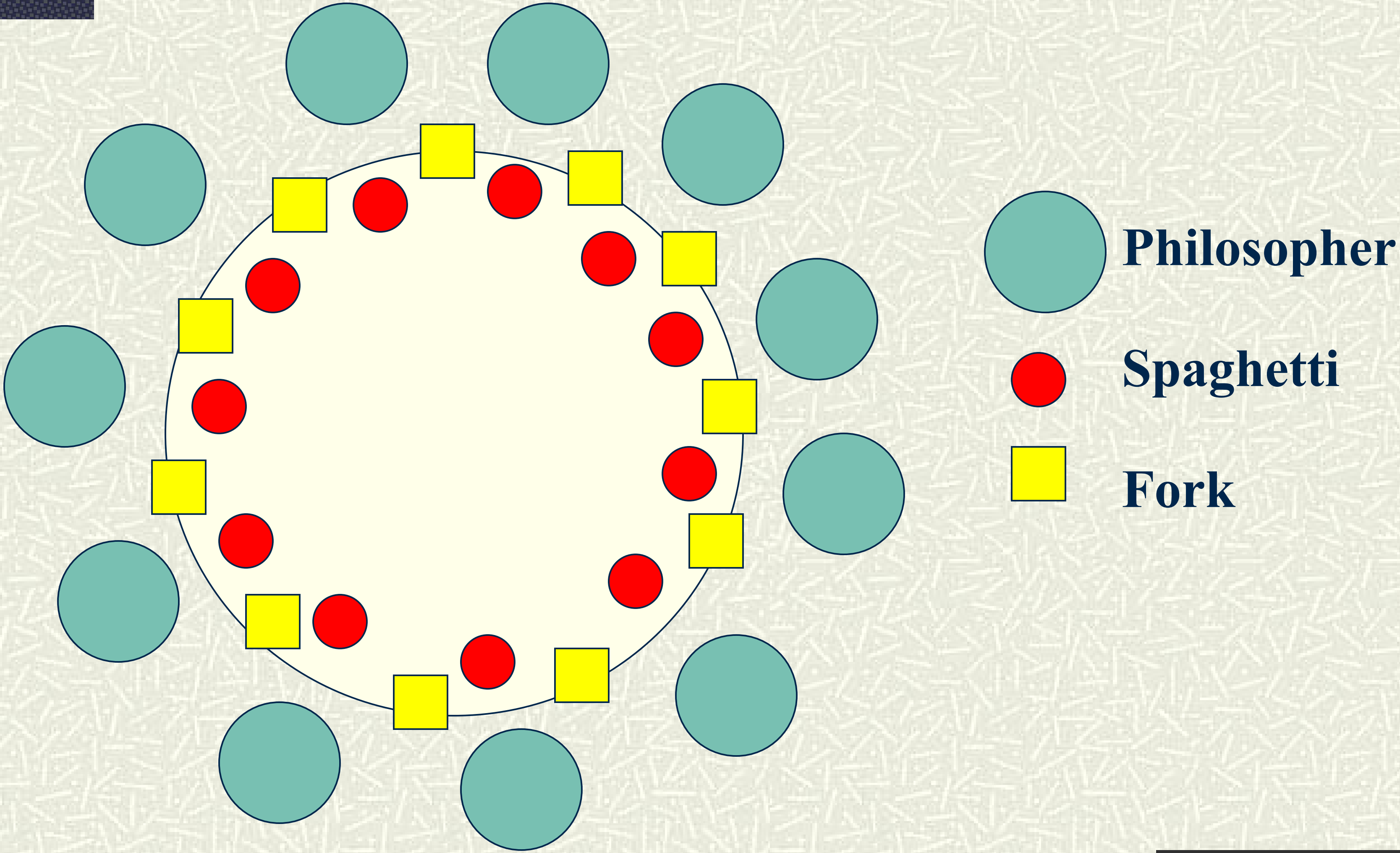
Ordering of Unlocking

- Since mutex_unlock does not force threads to wait, then the ordering of unlocking does not matter.

Dining Philosophers Problem

- N Philosophers are in a round table.
- There is a fork in each side of a plate that they will use to it spaghetti.
- They need two forks to eat the spaghetti.
- They will have to share the forks with their neighbors.

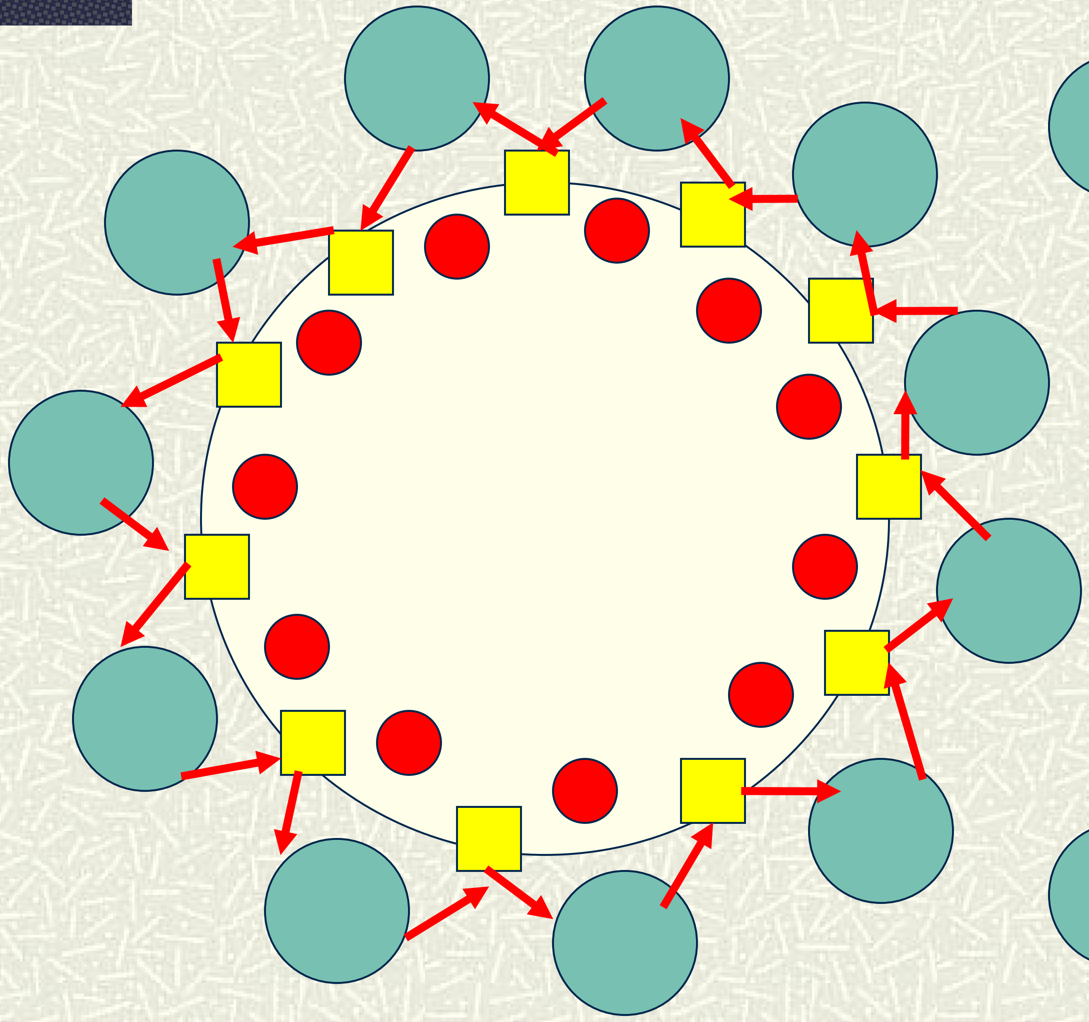
Dining Philosophers Problem

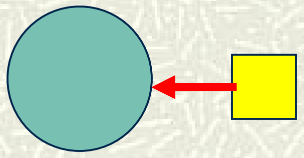


Dining Philosophers Problem

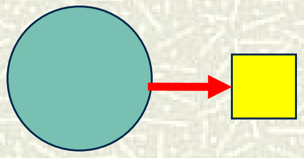
- Problem:
  - They may all decide to grab the fork in their right at the same time and they will not be able to proceed. This is a deadlock

Dining Philosophers Problem





Philosopher holds fork



Philosopher waits for fork

Dining Philosophers Problem (Unfixed Version)

```
const int NPHILOSOPHERS=10;
mutex_t fork_mutex[NPHILOSOPHERS];
pthread_t threadid[NPHILOSOPHERS];

void eat_spaghetti_thread(int i)
{
  while (i_am_hungry[i]) {
    mutex_lock(&fork_mutex[i]);
    mutex_lock(&fork_mutex[(i+1)%NPHILOSOPHERS);
    // Eat spaghetti
    chomp_chomp(i);
    mutex_unlock(&fork_mutex[i]);
    mutex_unlock(&fork_mutex[(i+1)%NPHILOSOPHERS);
  }
}
```

Dining Philosophers Problem (Unfixed Version)

```
main()
{
  for (int i = 0; i < NPHILOSOPHERS; i++) {
    mutex_init(&_fork_mutex[i]);
  }
  for (int i = 0; i < NPHILOSOPHERS; i++) {
    pthread_create(&threadid[i],eat_spaghetti_thread, i, NULL);
  }
  // Wait until they are done eating
  for (int i = 0; i < NPHILOSOPHERS; i++) {
    pthread_join(&threadid[i]);
  }
}
```

Dining Philosophers Problem (Fixed Version)

- The fork mutex have to be locked in order to prevent any deadlock.

Dining Philosophers Problem (Fixed Version)

```
void eat_spaghetti_thread(int philosopherNum)
{
  while (i_am_hungry[philosopherNum]) {
    int i = philosopherNum;
    int j = (philosopherNum+1)% NPHILOSOPHERS;
    if (i > j) { /*Swap i and j */
       int tmp=i; i=j; j=tmp;
    }
    // Lock lower order mutex first
    mutex_lock(&fork_mutex[i]);
    mutex_lock(&fork_mutex[j]);
    // Eat spaghetti
    chomp_chomp(philosopherNum);
    mutex_unlock(&fork_mutex[i]);
    mutex_unlock(&fork_mutex[j]);
  }
}
```

Condition Variables

- Condition Variables is a mechanism used for synchronization like semaphores.
- Condition variables and semaphores are equivalent in the sense that they can solve the same problems of synchronization
- Condition variables are more expressive than semaphores but using one vs. the other is a matter of taste.

Atomic Sections in Java

- In Java atomic sections are implemented using the “synchronized” keyword.
- No explicit calls to mutex_lock()/mutex_unlock() are needed.
- Any Java object can serve as a lock.
- Java implements synchronization in three ways:
  - Synchronized Class Method
  - Synchronized Instance Method
  - Synchronized Statement

Synchronized Class Method

- All the statements in the method become the atomic section (synchronized block) and the Class object is the lock.

```
class class_name {
    static synchronized type method_name1() {
     statement block
    }
 static synchronized type method_name2() {
     statement block
    }

}
```

- All static synchronized methods in the class are

Synchronized Instance Method

- All the statements in the method are part of the atomic section (synchronized block) and the instance object (this) is the lock.

```
class class_name {
    synchronized type method_name1() {
     statement block
    }
 synchronized type method_name2() {
     statement block
    }

}
```

Synchronized Statement

- All the statements inside the defined synchronized block become an atomic section.
- The object in the parenthesis is the lock.

```
class class_name {
    type method_name() {
     synchronized (object) {
          statement block
     }
    }
}
```

- All synchronized statements that share the same object will be mutually exclusive.

Condition Variables in Java

- Inside a synchronized block that uses an "***object***" as the lock, you can use
  - object.wait()
    - The caller will wait until another thread calls "notify" or "notifyAll" on the object
  - object.notify()
    - Wake up one thread waiting for object.wait.
  - object.notifyAll()
    - Wake up all threads waiting for object.wait.

Bounded Buffer in Java

- Implement a BoundedBuffer class to communicate producer and consumer threads using a circular buffer.
- Similar to the BoundedBuffer class with Semaphores and Condition Variables

Bounded Buffer in Java

```
BoundedBuffer.java:

class BoundedBuffer {
    int [] buffer;
    int tail;
    int head;
    int n;
    int max;

    // Used to synchronize bounded buffer
    // as mutex and cond var.
    final Object monitor = new Object();

    public BoundedBuffer(int max) {
        buffer = new int[max];
        tail = 0;
        head = 0;
        n = 0;
        this.max = max;
    }
```

BoundedBuffer.enqueue(int v)

```
public void enqueue(int v) {
  synchronized(monitor) {
      // Wait if buffer is full
      while (n==max) {
          try {
              monitor.wait();
          }
          catch (InterruptedException e) {}
      }

      buffer[tail] = v;
      tail = (tail+1)%max;
      n++;

      // Wakeup any thread waiting in dequeue
      monitor.notifyAll();
  }
}
```

BoundedBuffer.dequeue()

```
    public int dequeue() {
      int v;
      synchronized(monitor) {
          // Wait if buffer is empty
          while (n==0) {
               try {
                    monitor.wait();
               }
               catch (InterruptedException e) {}
          }

          v = buffer[head];
          head = (head+1)%max;
          n--;

          // Wakeup any thread waiting in dequeue
          monitor.notifyAll();
      }
      return v;
    }
}
```

Bounded Buffer Consumer Thread

```
Consumer.java:
class Consumer extends Thread {
    BoundedBuffer bbuffer;

    Consumer(BoundedBuffer bbuffer) {
        this.bbuffer = bbuffer;
    }

    // This method is called when the thread runs
    public void run() {
       System.out.println("Consumer Thread Running");
       int last = -1;
       int i = 0;
       while (true) {
           i = bbuffer.dequeue();
           System.out.println("consume("+i+")");
           if ( last +1 != i) {
            System.out.println("**** Error: last="+last+" i="+i);
           }
           last = i;
        }
    }
```

Bounded Buffer Producer Thread

```
Producer.java:
class Producer extends Thread {
   BoundedBuffer bbuffer;

   public Producer() {
        bbuffer = new BoundedBuffer(10);
   }

   public void run() {
        System.out.println("Producer Thread Running");

        int i = 0;
        // Start consumer
        Consumer consumer = new Consumer(bbuffer);
        consumer.start();

        while (true) {
                System.out.println("produce("+i+")");
            bbuffer.enqueue(i);
            i++;
        }
   }
```

Bounded Buffer Producer Main

```
    public static void main(String [ ] args) {
       // Start producer. Producer will start consumer
       Thread thread = new Producer();
       thread.start();
    }
}
```

Bounded Buffer Output

```
bash-4.0$ javac *.java
bash-4.0$ java Producer
Producer Thread Running
Consumer Thread Running
produce(0)
produce(1)
…
produce(9)
produce(10)
consume(0)
consume(1)
consume(2)
consume(3)
…
consume(26731)
produce(26732)
consume(26732)
produce(26733)
…
```

User and System Time in MT programs

- Command “time” gives
  - User Time = Time spent in user mode
  - System Time = Time spent in Kernel mode
  - Real Time = wall clock time.
- In a single processor machine
  - User time + system time < Real time
  - Why? Overhead of other programs running in the same machine. Also cpu waits for the I/O to finish.

User and System Time in MT programs

- In a multi processor machine
  - User time + system time < N* Real time
  - Where N is the number of processors
- If your program is using the N processors at 100% then
  - User time + System time = N* Real time.

or

(User Time+ System time)/Real Time = N

Types of Computer Systems

- **Parallel Systems**
  - Have more than 1 CPU in the same computer
  - SMP – Symmetric Multiprocessing. Each CPU runs a copy of the Operating System
  - A task that takes T seconds to complete may take ideally T/N seconds.
  - However, some tasks that have to be executed serially may not take full advantage of parallel computers. Some other tasks can be parallelized very effectively.
  - They are expensive. The cost is $O(n^2)$ with the number of processors. Due to communication hardware that allows interaction between processors.

Types of Computer Systems

- **Clusters**
  - Collection of inexpensive computers connected through a fast network.
  - Alternative to parallel computers.
  - Cost is linear O(n) with the number of nodes used. More robust than parallel computers since they can be serviced while running.
  - Still the communication across nodes is slower than the communication across processors in parallel computers.

Types of Computer Systems

- **Clusters (cont.)**
  - Programs may need to be rewritten to take advantage of clusters since each cluster will run different processes.
  - Some Database servers may run better in parallel computers.
  - The best example of a cluster is Google that has clusters with about 250,000 computers distributed in different locations. Google searches are resolved by multiple machines running in parallel.
  - Another example are rendering farms used to create computer animated movies like “Finding Nemo”, “Shrek” etc.

Types of Computer Systems

- **Grid**
  - A collection of inexpensive computers across the internet running the same application to accomplish a collective task.
  - People donate idle time of their computers to be used in the grid.
  - Example: SETI@home (Search for extra-terrestrial intelligence) + You can download a program that will analyze radio telescope data looking for "intelligent" patterns.

Virtual Machines (VM)

- A VM is a virtual machine running in a physical computer.
- More than one VM can run simultaneously in the same machine.
- VMs work well because machines are now too fast and underused.
- It allows to “snapshot” in a file of an OS together with the applications it has installed.
- You can save the VM in disk and duplicate it.

VM Applications

- Hosting multiple server environments in the same machine.
- Using multiple OS in the same machine.
- Testing programs in multiple OS versions and environments without the need of multiple machines.
- Simplifies administration of systems since an administrator can install the Apps in one VM and then install the image in multiple machines.

VM Software

- VMWare
- Virtual Box
- Citrix

Midterm Review

- What is an Operating System
- The UNIX File System
  - Users and Groups
  - Disk Organization
  - I-nodes, Soft/Hard Links
  - File permissions and ownership
  - Permission bits

Midterm Review

- UNIX commands and Shell Scripting
  - ls, mkdir, cp, mv, rm, grep, man, where, which head, tail, awk, sed, find
  - Shell scripting and examples
- Program Structure
  - Memory Sections (text, data, bss, heap, stack)
  - Executable File Formats
  - Steps to build a program
  - Steps to load a program
  - Static and Shared Libraries

Midterm Review

- Using a debugger
  - Running gdb
  - Breakpoints
  - Next/step/run
  - Print
  - Where
  - Debugging a crashed program

Midterm Review

- UNIX System Programming
  - The Shell project
  - Lex and yacc
  - Shell Grammar
  - Open File Table
  - open(), close, fork(), exec(), dup, dup2, pipes
  - Wildcards

Midterm Review

- Computer Architecture Review
  - Kernel and User Mode
  - Interrupts
  - Poling
  - Interrupt Vector
  - Types of Interrupts
  - System Calls
  - System Call and Security Enforcement
  - System Calls and Interrupts Example

Midterm Review

- Processes
  - States of a Process
  - I/O Bound and CPU Bound Processes
  - Process Table
  - Preemptive and non-preemptive scheduling
  - Context Switch Overhead
  - Multi-level Feedback Queue Scheduling

Midterm Review

- Processes (cont.)
  - Process ID,
  - Arguments,
  - Environment Variables,
  - Current Directory,
  - Stdin/Stdout/Stderr

Midterm Review

- Threads
  - Comparison of threads and processes
  - Critical Sections
  - Mutex locks
  - Spin locks
  - Semaphores
  - Synchronized list
  - Bounded Buffer
  - Read/Write Lock
  - Deadlock and Starvation
  - Deadlock Prevention

Midterm Review

- To Study
  - Class Slides
  - Final Review
  - Projects

Internet Review and Socket Programming

History of the Internet

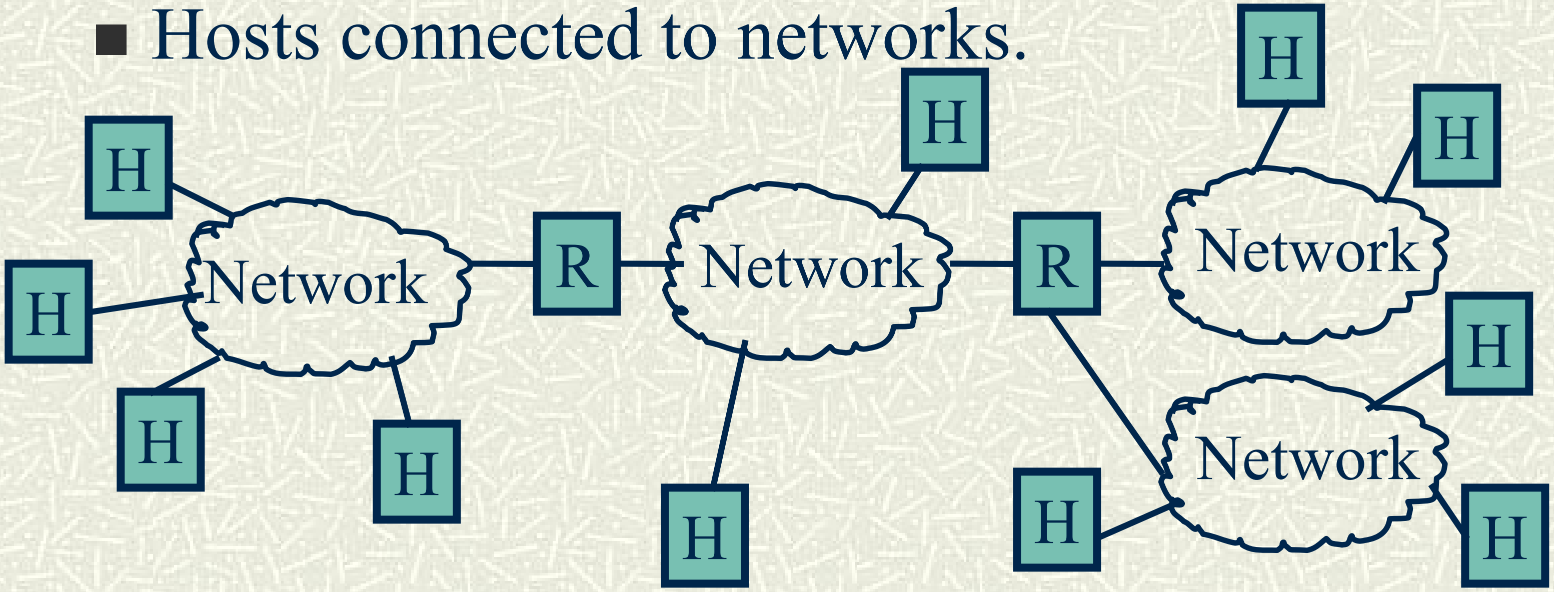
- In the late 1960s the Department of Defense Advance Research Project Agency (DARPA) created a nationwide network to allow computer access to the different research centers.
- The alternative was to give a computer to each center/university but this was –very- expensive.
- Research in the Internet continued in the 70s and 80s
- The Internet became a commercial success in the 90s.

History of the Internet

- The Internet has been doubling in size every nine to twelve months.
- Some people attribute the increase in productivity in the last 10 years to the existence of the Internet. People produce more in less time.

Internet Architecture

- The Internet is a collection of
  - Networks
  - Routers interconnecting networks
  - Hosts connected to networks.

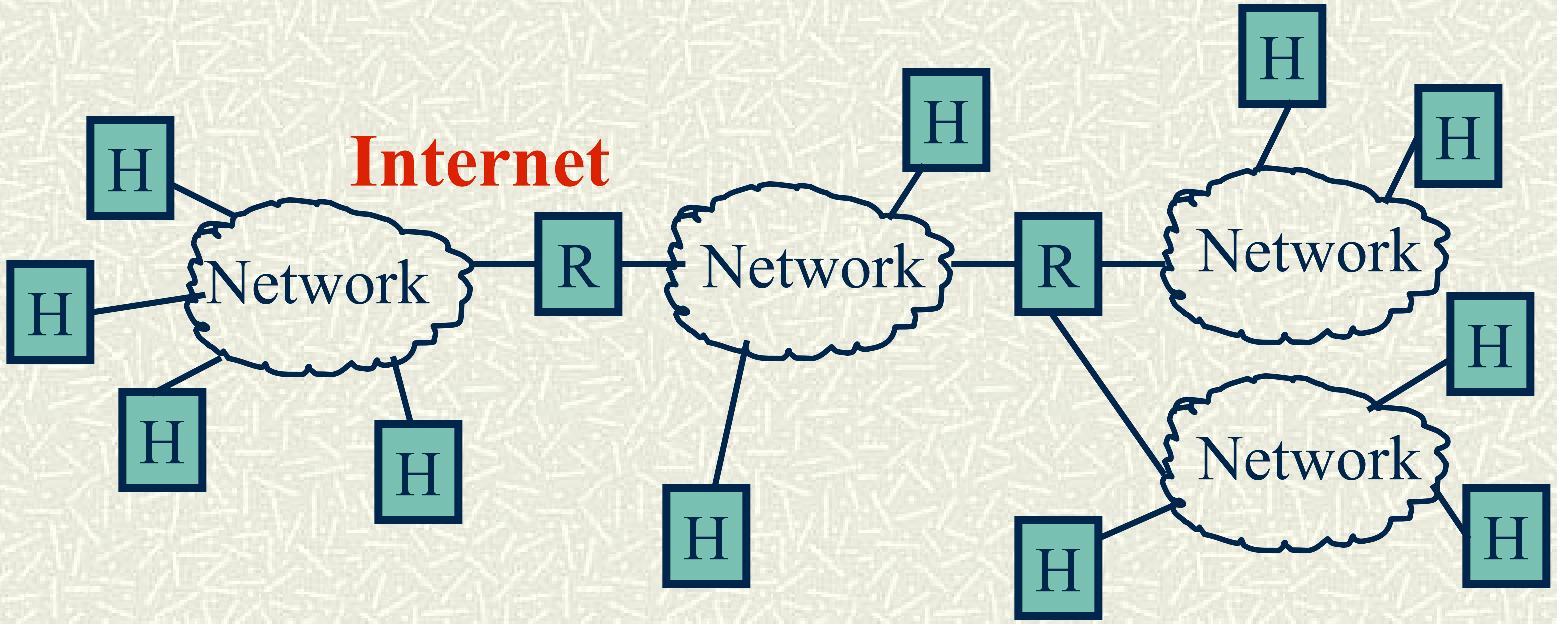


Internet Architecture

- The networks may be implemented using different kinds of hardware: Ethernet, Token Ring, Serial Line, Apple Talk, etc.
- The goal of the Internet is to hide all this heterogeneity to the user and user programs.

Internet Architecture

- The internet is a virtual network with its own addressing and name scheme.



Internet Layering

- It reflects the layering used by the TCP/IP protocols
- Closer to reality than ISO-& Layering

Application	- Individual Application Programs (HTTP)
Transport	- Program to Program (TCP and UDP)
Internet	- Packet Forwarding. Machine to Machine. (IP)
Network Interface	- Local Area Network (Ethernet, RS232, etc)
Physical	- Basic Network Hardware

IP Addressing

- It is an abstraction to hide the network internals.
- It is independent from hardware addressing.
- IP addresses are used for all communications.
- IP addresses use 32 bits.
- There is a unique value for each host

IP Addressing

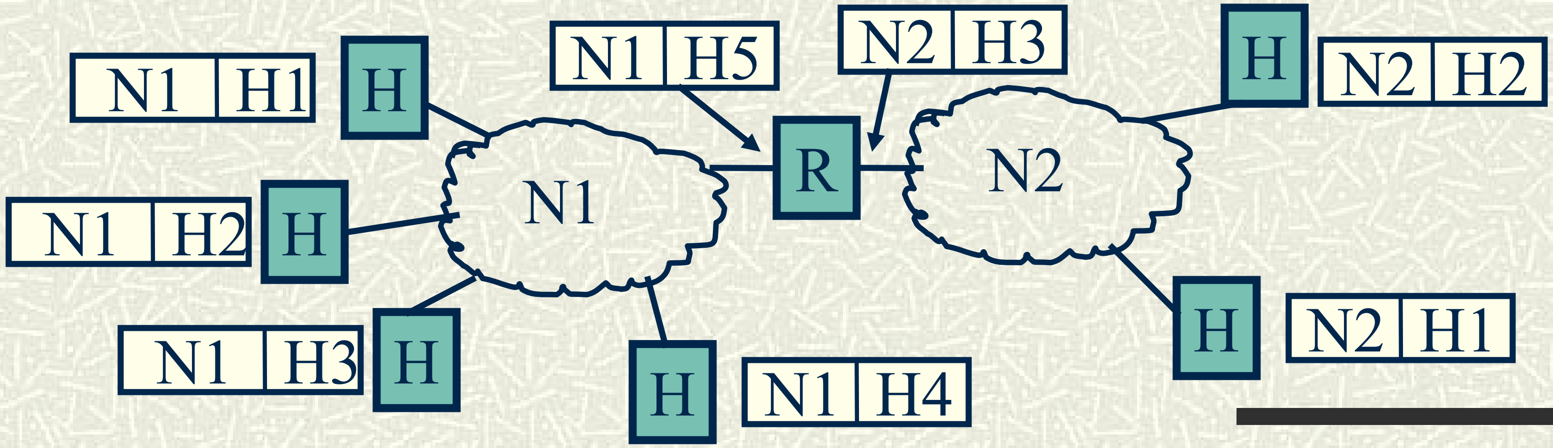
- Important:

  An IP Address does not specify a specific computer. Instead, each IP address identifies a connection between a computer and a network.
- An IP address identifies a network interface.
- A computer with multiple network connections (like a router) must be assigned one IP address for each connection.

IP Addressing

- It has two parts:
  - The prefix identifies a network
  - The suffix identifies the host in that network.

Network Number	Host Number



IP Addressing

- A global authority assigns a unique prefix for the network.
- A local administrator assigns a unique prefix to the hosts.
- The number of bits assigned to the prefix and suffix is variable depending on the size of the number of hosts in each network.
- The subnet mask is a parameter in the interface that tells the number of bits used

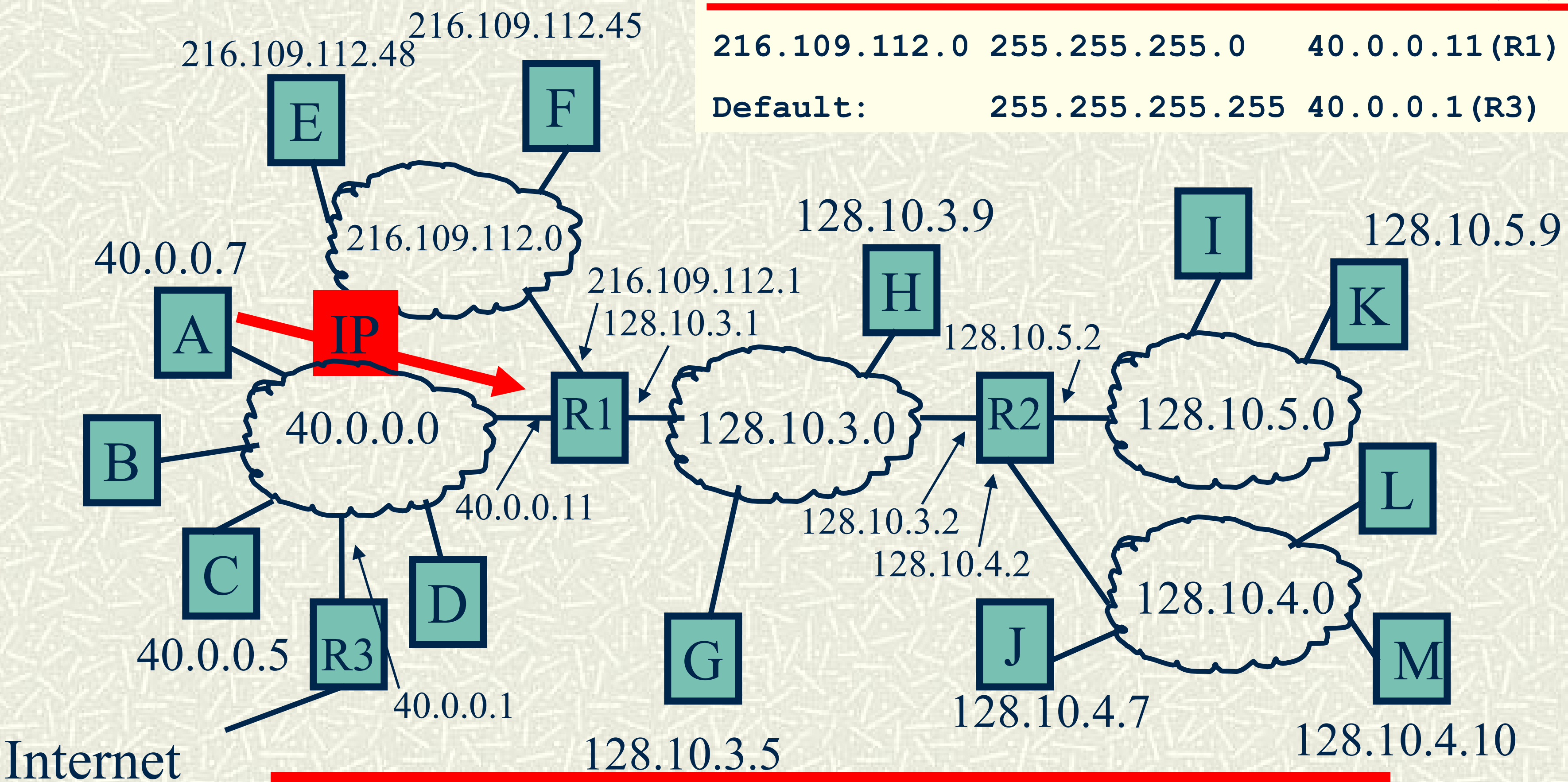
Routing

- The routing table gives the next router necessary to reach the destination network.
- The source of the table information can be:
  - Manual:
    - By hand
    - Small networks
    - OK if routes never change
  - Automatic
    - Software creates/updates the routing table using information from neighboring routers.
    - It is needed for lager nets
    - It changes routes if failure.

IP packet from A to M

A: Routing Table

Target Net	Net/Subnet Mask	Next Hop
40.0.0.0	255.0.0	Directly
128.10.3.0	255.255.255.0	40.0.0.11(R1)
128.10.5.0	255.255.255.0	40.0.0.11(R1)
128.10.4.0	255.255.255.0	40.0.0.11(R1)
216.109.112.0	255.255.255.0	40.0.0.11(R1)
Default:	255.255.255.255	40.0.0.1(R3)

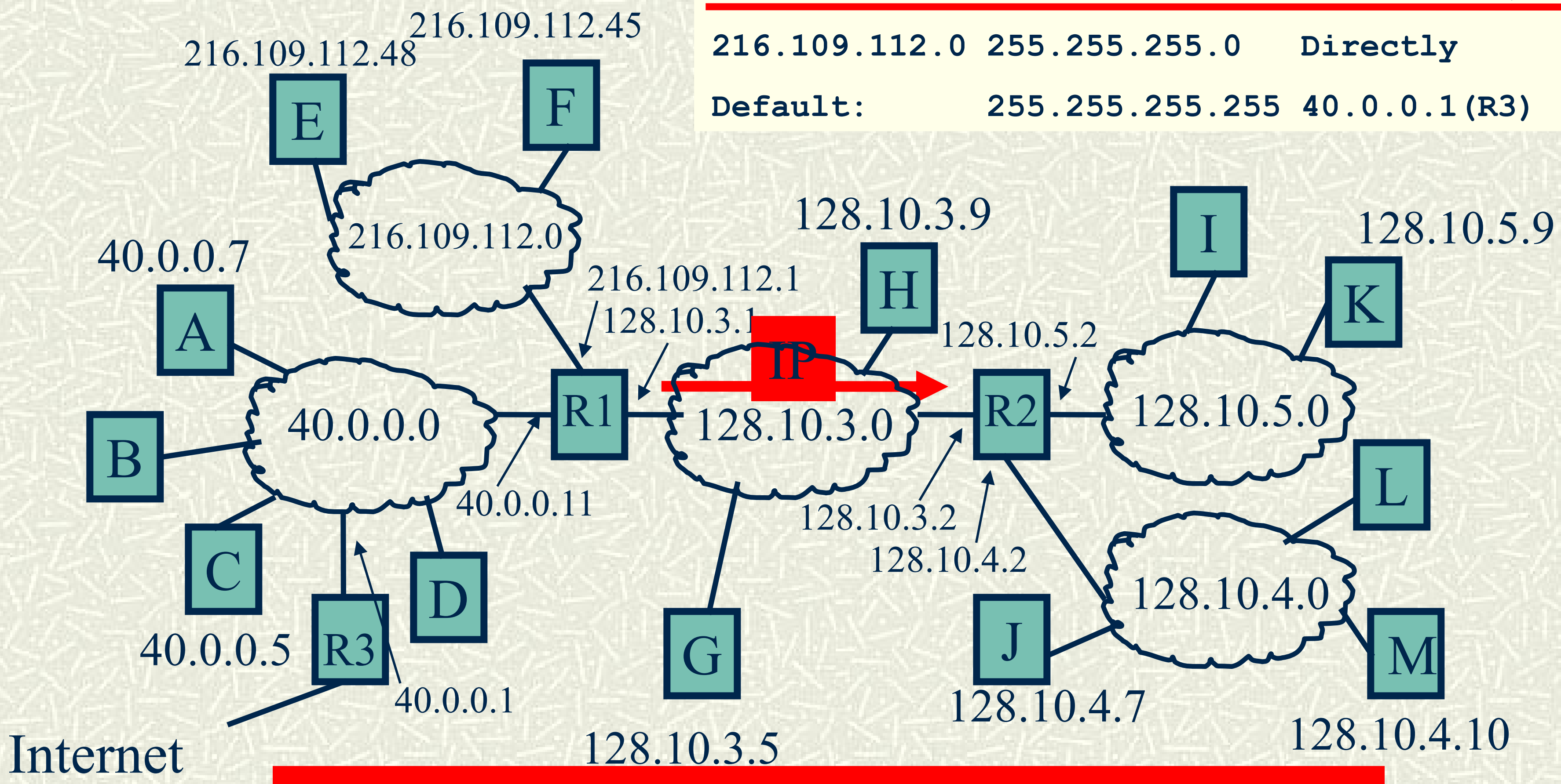


IP: $E_{src}=E_a$, $E_{dst}=E_{R1}$, $IP_{src}=A$, $IP_{dst}=M$

IP packet from A to M

R1: Routing Table

Target Net	Net/Subnet Mask	Next Hop
40.0.0.0	255.0.0	Directly
128.10.3.0	255.255.255.0	Directly
128.10.5.0	255.255.255.0	128.10.3.2(R2)
128.10.4.0	255.255.255.0	128.10.3.2(R2)
216.109.112.0	255.255.255.0	Directly
Default:	255.255.255.255	40.0.0.1(R3)

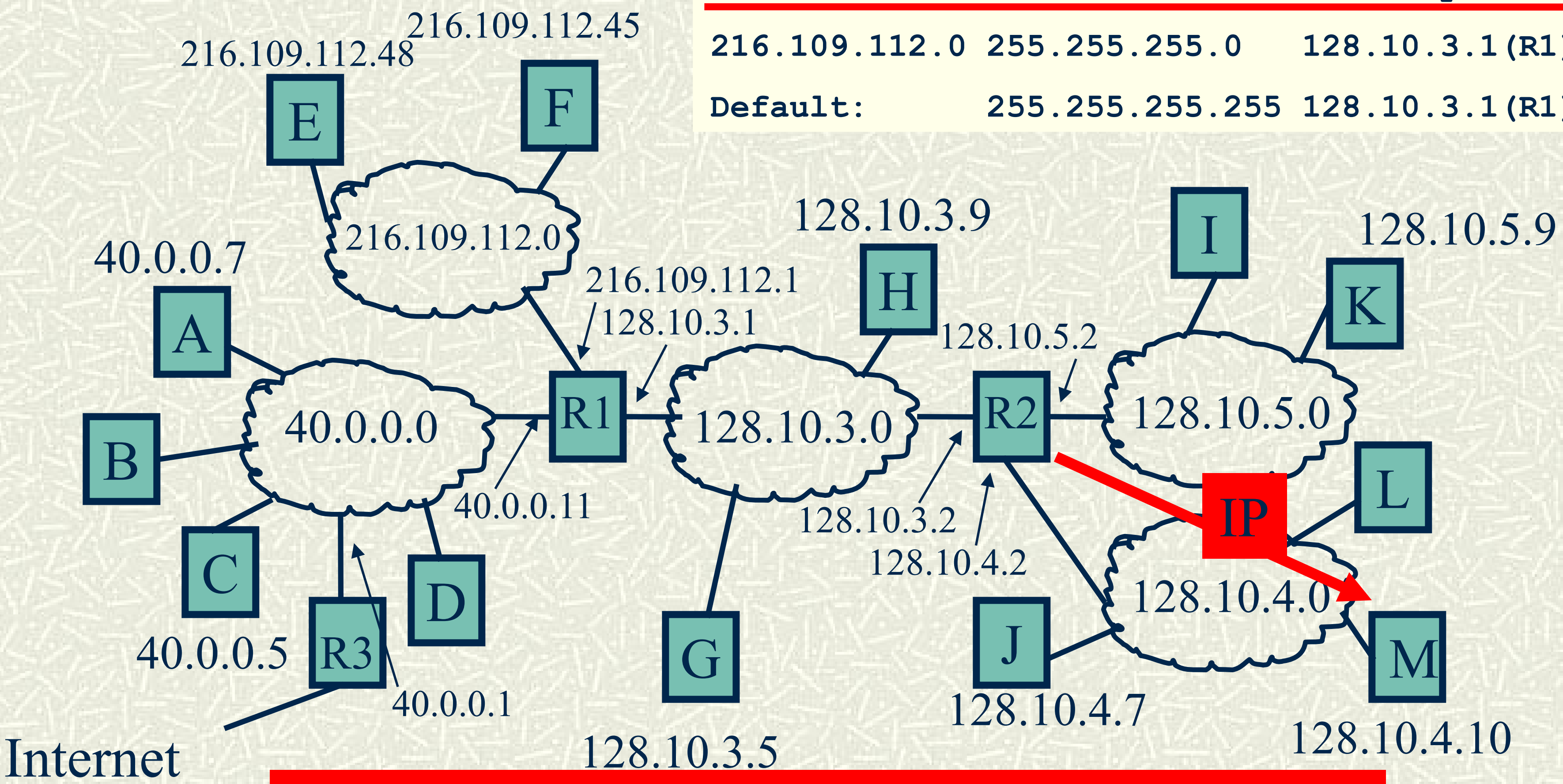


IP: $E_{src}=E_{R1}$, $E_{dst}=E_{R2}$, $IP_{src}=A$, $IP_{dst}=M$

IP packet from A to M

R2: Routing Table

Target Net	Net/Subnet Mask	Next Hop
40.0.0.0	255.0.0	128.10.3.1(R1)
128.10.3.0	255.255.255.0	Directly
128.10.5.0	255.255.255.0	Directly
128.10.4.0	255.255.255.0	Directly
216.109.112.0	255.255.255.0	128.10.3.1(R1)
Default:	255.255.255.255	128.10.3.1(R1)



IP: E_{src}=E_{R2}, E_{dst}=E_{M}, IP_{src}=A, IP_{dst}=M

Addresses in IP packet

- The IP source and destination address of the packet is the same during the transit of the packet.
- The hardware source and destination address will be different every time the packet is forwarded.
- The source host or some of the routers also may require ARP requests if the hardware destination address is not in the ARP cache.

ARP Address Resolution Protocol

- When it is time for the router or host to deliver a packet directly, it is necessary to convert the IP address to a hardware address.
- For example in an Ethernet LAN , the target IP address in the IP packet has to be translated to the Ethernet address of the destination machine.
- ARP does this translation.

ARP Address Resolution Protocol

- ARP Input and Output:
  - Input: IP address C of computer in locally connected network N
  - Output: Ethernet address for C.
- ARP keeps bindings (IPAddr, EtherAddr) in a table called ARP table or ARP cache.
- ARP builds the table as needed.

ARP Command

```
C:\Users\owner>arp -a
Interface: 10.184.105.105 --- 0xd
  Internet Address      Physical Address      Type
  10.184.96.1           00-24-c4-c0-fe-c0     dynamic
  10.184.111.255        ff-ff-ff-ff-ff-ff     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

DNS: Domain Name Server

- Humans prefer to use computer names instead of IP addresses.
- Example: www.yahoo.com instead of 204.71.200.68
- Before DNS the mappings name to IP address where stored in a file /etc/hosts
- The Net administrators used to exchange updates in the /etc/hosts file.
- This solution was not scalable..

DNS: Domain Name Server

- One host name may map to multiple IP addresses. Why? One host will have multiple IP addresses if it has multiple network interfaces.

DNS: Domain Name Server

- DNS is a service that translates host names to IP addresses.
- DNS uses a distributed lookup algorithm and contacts as many servers as necessary.
- Host names are divided in domains: host.dom2.dom1.dom0.
- Example: ector.cs.purdue.edu with the most general domain at the right.

DHCP – Dynamic Host Configuration Protocol

- Allows connecting computers to the Internet without the need of an administration.
- Before DHCP, an administrator had to manually configure the following parameters to add a computer to the Internet:
  - The local IP address – Current address
  - The subnet mask – Used to send packets to hosts in same LAN
  - The default router – Used to send packets to hosts outside the LAN
  - The default DNS server – Used to convert names to IP addresses.
- In UNIX the command used to set these parameters is ifconfig. In Windows is ipconfig or the Control Panel.

Transport Protocols

- Two transport protocols available in the TCP/IP family
  - UDP – User Datagram Protocol
  - TCP – Transmission Control Protocol

UDP- User Datagram Protocol

- Unreliable Transfer. Applications will need to implement their own reliability if necessary.
- Minimal overhead in both computation and communication.
- It is best for LAN applications
- Connectionless – No initial connection necessary. No state in both ends

UDP- User Datagram Protocol

- Message Oriented
- Each message is encapsulated in an IP datagram.
- Size of message is restricted by the size of the MTU of the directly connected network. (Maximum Transfer Unit =1500bytes for Ethernet networks)
- The UDP header has ports that identify
  - Source application (Source Port)
  - Destination application (Destination Port)

TCP – Transmission Control Protocol

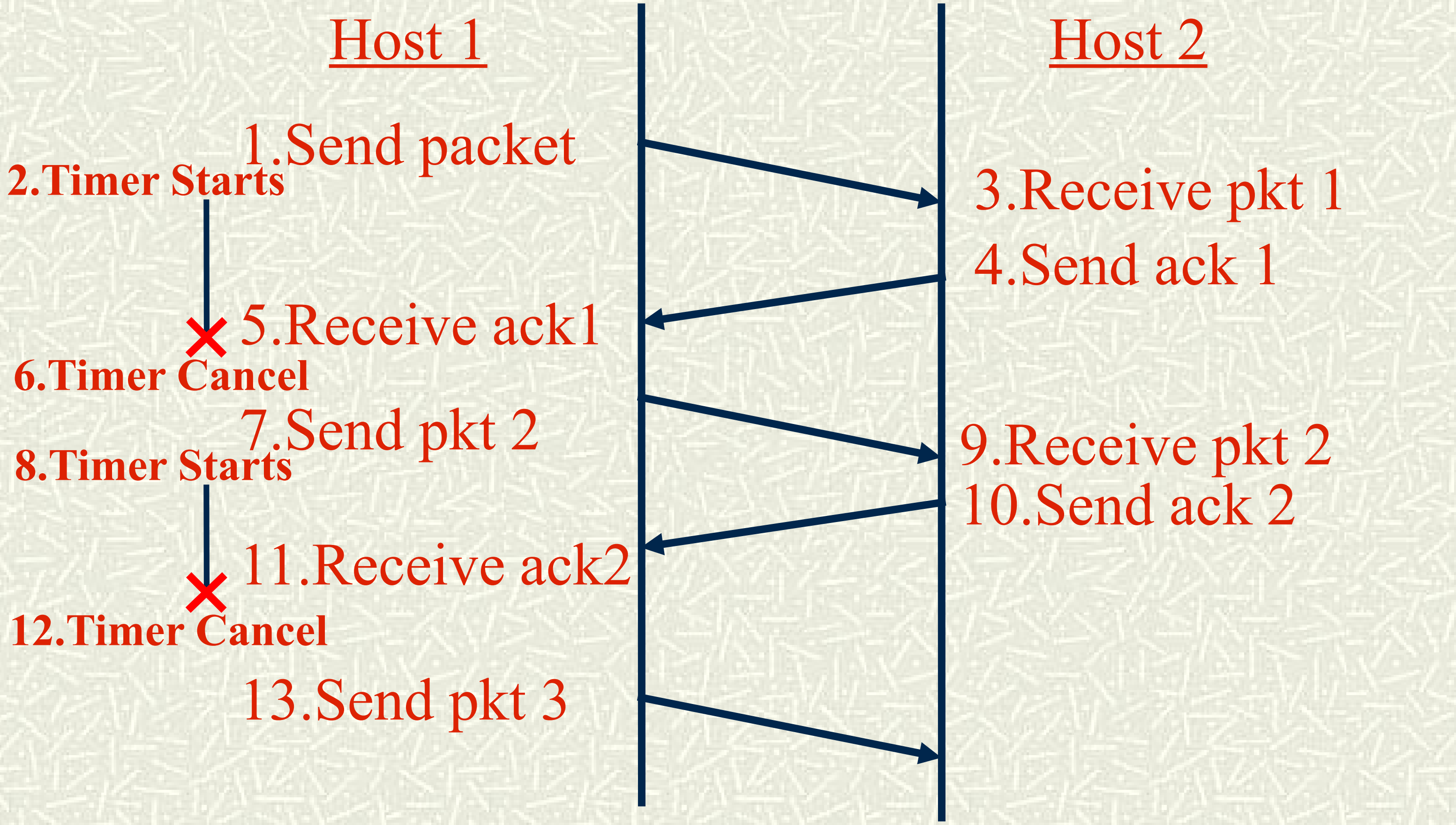
- It is the major transport protocol used in the Internet
- It is:
  - Reliable – It uses acknowledgement and retransmission to accomplish reliability
  - Connection-Oriented - An initial connection is required. Both end points keep state about the connection.
  - Full-Duplex – Communication can happen in both ways simultaneously.
  - Stream Interface – Transfer of bytes look like writing/reading to a file.

TCP Reliability

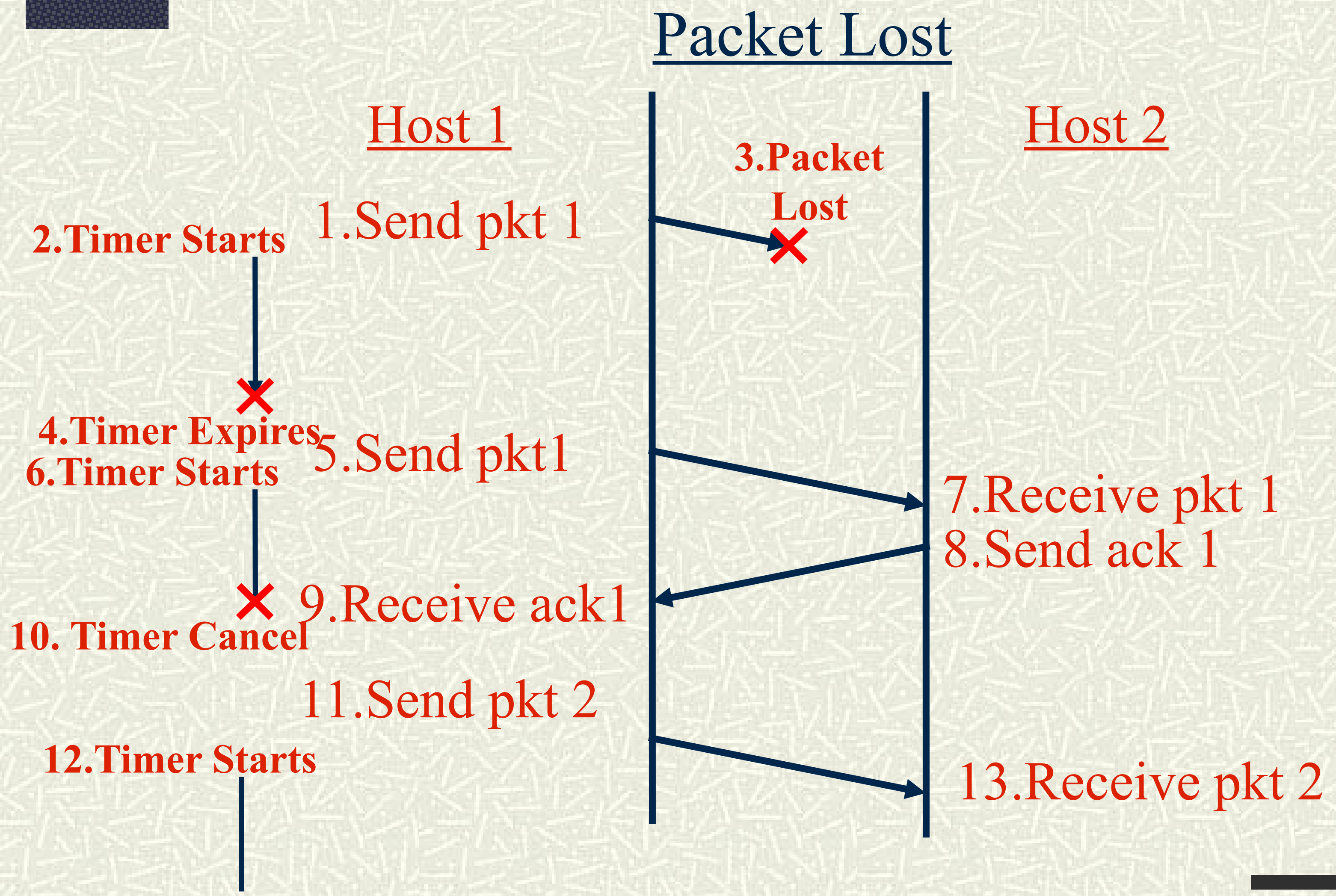
- How does TCP achieves reliability?
- It uses Acknowledgments and Retransmissions
- Acknowledgement-
  - The receiver sends an acknowledgement when the data arrives.
- Retransmission
  - The sender starts a timer whenever the message is transmitted
  - If the timer expires before the acknowledgement arrives, the sender retransmits the message.

TCP Reliability

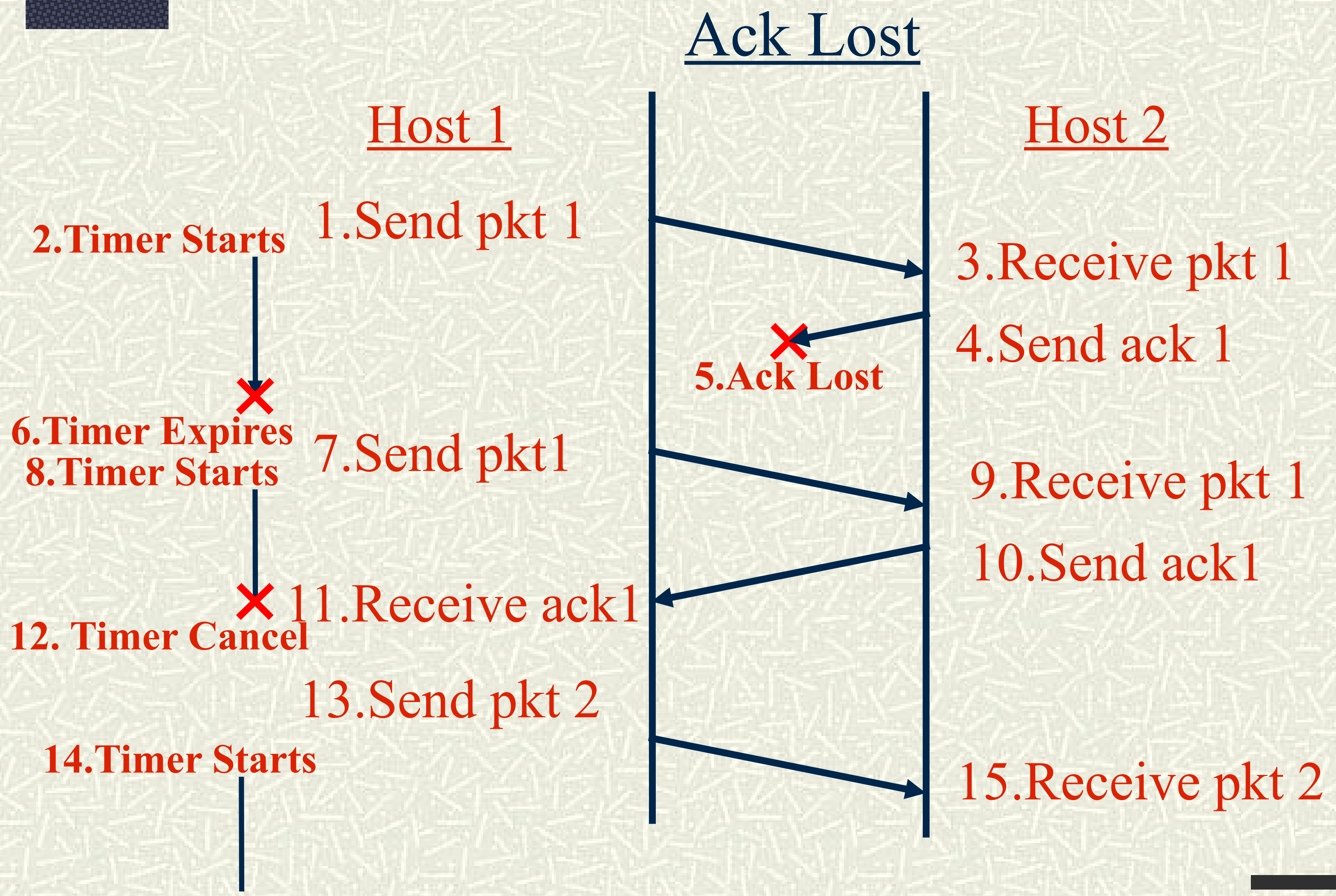
Normal Exchange



TCP Reliability



TCP Reliability



TCP Summary of Features

- *1. Adaptive Retransmission*
  - The retransmission timer is set to RTT+4*RTTVAR where RTT is estimated. This allows TCP work in slow and fast networks.
- *2. Cumulative Acknowledgments*
  - An acknowledgment is for all the bytes received so far without holes and not for every packet received.
- *3. Fast Retransmission*
  - It is a heuristic where a duplicated acknowledgment for the same sequence is signal of a packet lost. The data is retransmitted before the timer expires.

TCP Summary of Features

- ***4. Flow Control***
  - It slows down the sender if the receiver is running out of buffer space. The window (receiver's buffer size) is sent in every acknowledgment.
- ***5. Congestion Control***
  - For TCP a lost packet is signal of congestion.
  - Instead of aggressively retransmit, it will slow down the retransmission. It will use first ***"Slow Start"*** and then a ***"Congestion Avoidance"*** where the window of retransmitted data is reduced in size.

TCP Summary of Features

- ***6. Reliable Connection and Shutdown***
  - TCP uses a Three way Handshake to close connections.
  - Three packets are enough to make sure that lost packets or host crashes will not interfere future connections.

When to Use UDP or TCP

- If you need reliable communication in your application use TCP.
- Only use UDP in the following cases:
  - Broadcasting: that is the computer needs to reach all or part of the computers in the local network.
    - Example: Find out of the existence of a server (Example DHCP, or finding a printer).
    - Multicasting data to several machines simultaneously.
  - Real Time Data: Applications where packets arriving on time with minimum delay is more important than reliability where retransmission can add to the delay.
    - Example: Voice over IP, teleconferencing.

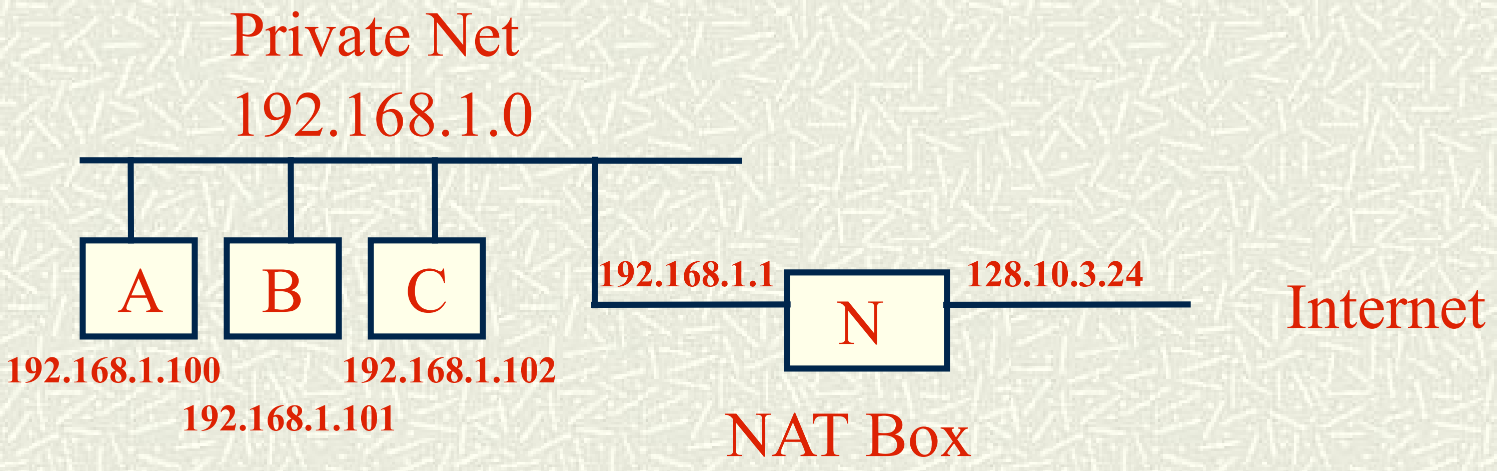
NAT: Network Address Translation

- *Network Address Translation (NAT)* is used when you want to connect multiple computers to the Internet using a single IP address.
- The NAT software can run on a computer or specialized device (NAT box) that has two network interfaces: one connected to the private network and the other one to the Internet.

NAT: Network Address Translation

- As a side effect, NAT provides protection.
- Packets will be allowed into the private network only if they belong to a connection that started by a machine in the private network.
- A NAT box is also called a ***Firewall.***
- NAT also mitigates the problem of running out of Assigned Network Numbers.
- Potentially you could have another Internet behind a NAT box.

NAT: Network Address Translation



From the point of view of the Internet, all computers in the private network have the address 128.10.3.24

The NAT box is also a DHCP server that assigns IP addresses and it is the default router.

NAT: Network Address Translation

- A TCP connection is defined uniquely in the entire Internet by four values:

  <src-ip-addr, src-port, dest-ip-addr, dest-port>

- The NAT box will work as follows:

1. The machines in the private network use the NAT box as the default router.
2. When a TCP packet with header

   <IPsrc, PORTsrc, IPdest, PORTdest>

   goes from the private network to the Internet, the NAT box will change the header to

   <IPnat, PORTrand, IPdest, PORTdest>

   Where ***IPnat*** is the shared IP address and ***PORTrand*** is a random unused port in the NAT box.

NAT: Network Address Translation

3. The NAT box will also add the NAT mapping to the NAT table

   (PORTrand, IPsrc, PORTsrc)

4. When a packet

   < ***IPdest, PORTdest , IPnat, PORTrand***>

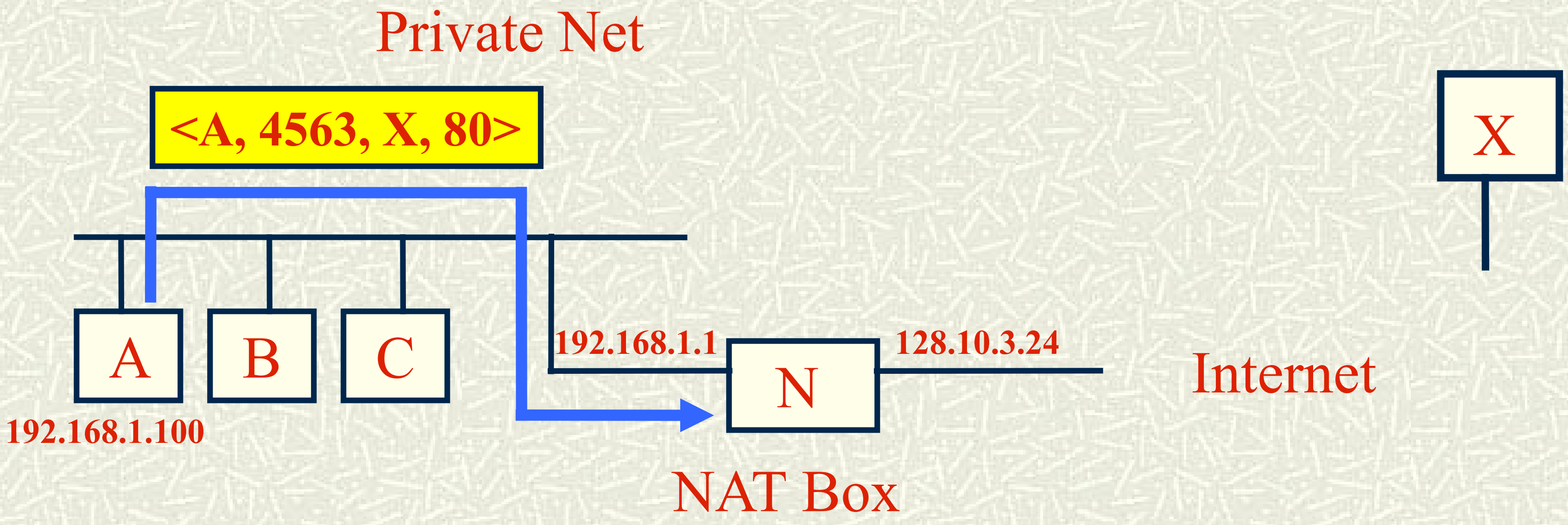
   comes from the Internet to the NAT box, the NAT box will lookup ***PORTrand*** in the NAT table and it will change the header to

   < ***IPdest, PORTdest , IPsrc, PORTsrc***>

   And it will forward the packet to ***IPsrc.***

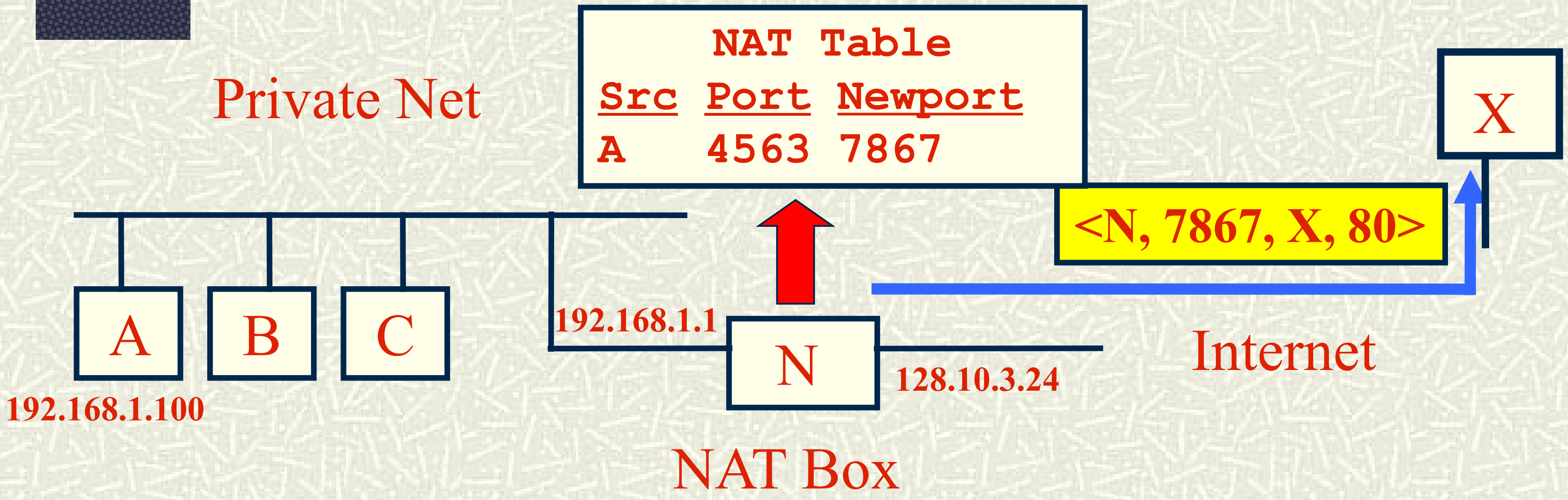
5. A similar translation is done for UDP packets.

NAT Example



1. Computer A wants to establish a TCP connection with an HTTP server X in the Internet.
2. The NAT box is the default router for A so A sends the TCP packet to the NAT box.

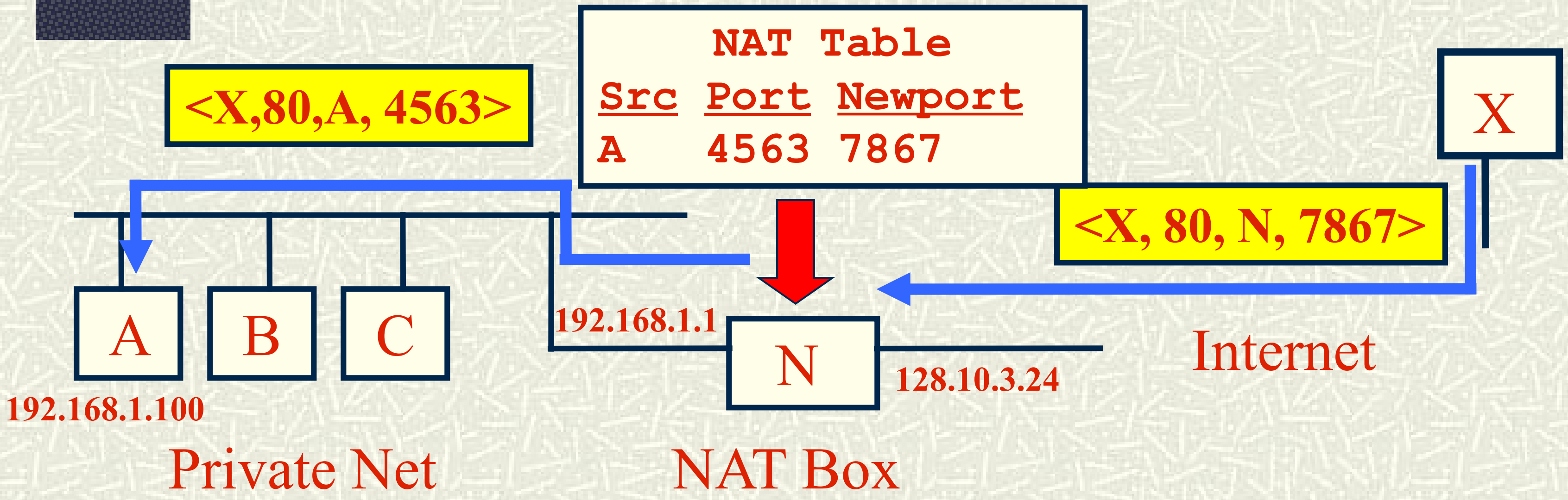
NAT Example



3. The NAT box chooses an unused random port (7867) and substitutes the source port in the packet as well as the source address with its own IP address. The new packet is sent to X.

4. The old port (4563), old source address (A), and new port are added to the NAT table for use when packets of the same connection come back.

NAT Example



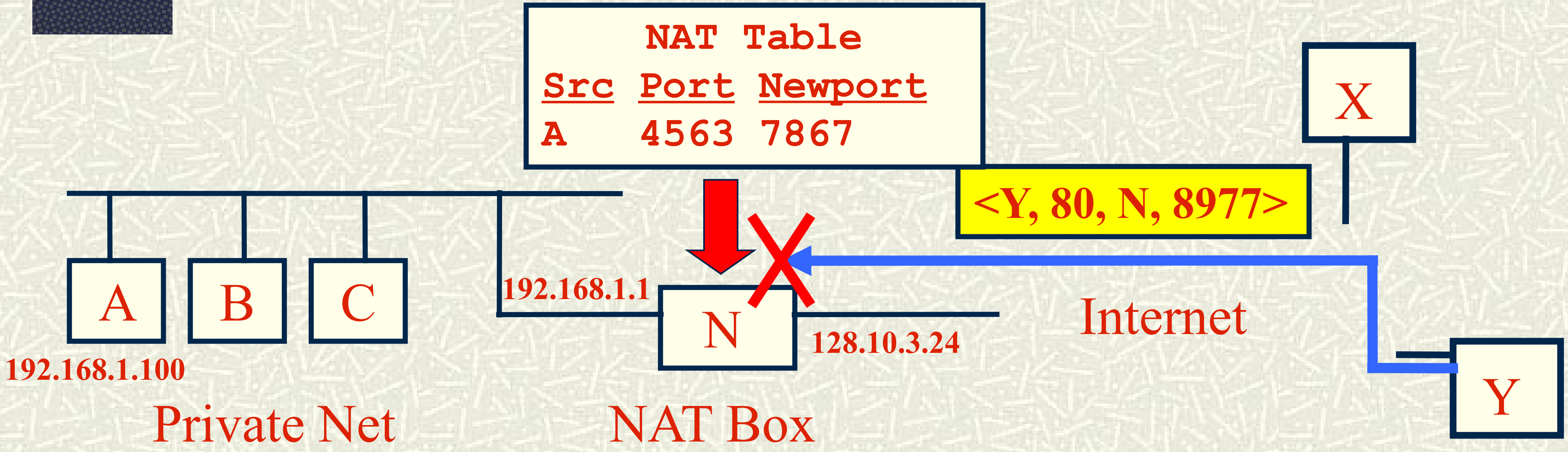
5. When a packet comes back from X, the NAT box will lookup the destination port (7867) in the NAT table and it will substitute the destination address and destination port with the original values.

6. The packet is forwarded to A. Everything will be transparent for both X and A. They will never know that a translation took place. The same mapping will be used for all packets of the same connection until the connections closed or the entry times out..

NAT and Firewalls

- The NAT box will pass into the private network only the packets that belong to a connection that started from the inside the private network.
- This is similar to the recommendation “Never give your credit card through the phone if you did not start the call”.
- If a packet is received that does not have a mapping in the NAT table, the packet is discarded.

NAT and Firewalls

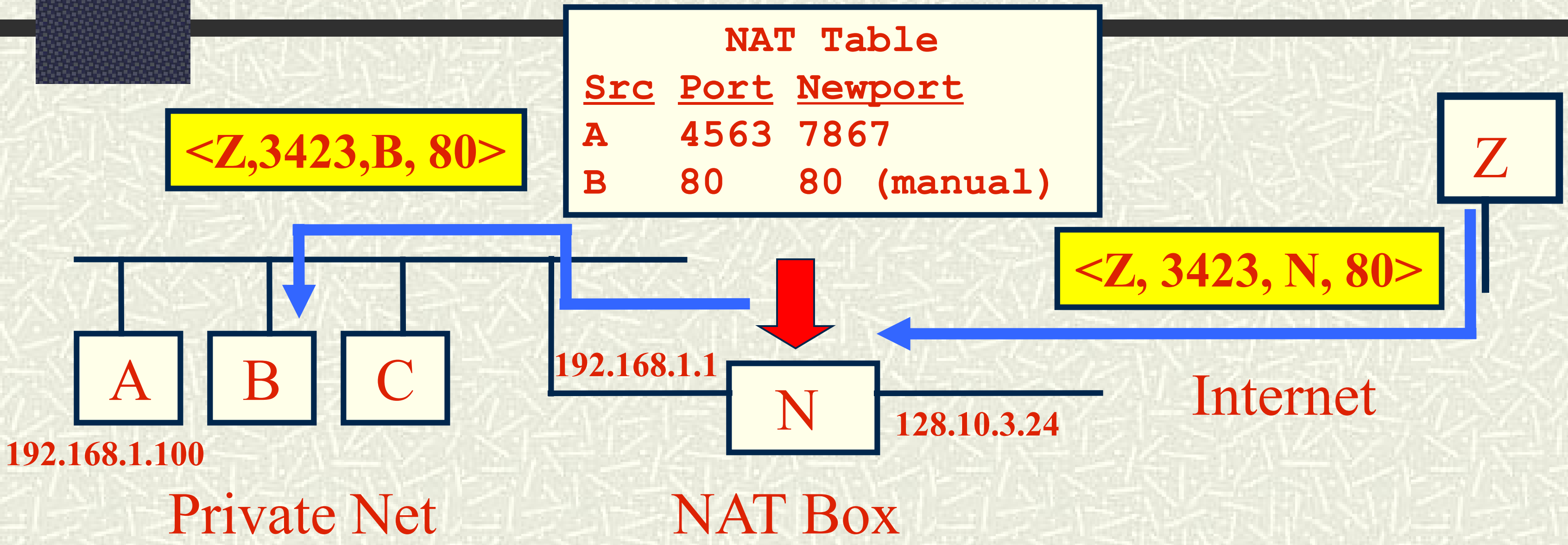


1. If a packet arrives to the NAT box that does not have an existing entry in the NAT table, the packet will be discarded by the NAT box.
2. This protects the private network from hacker attacks. It is recommended to have a NAT box/Firewall even when you have one computer connected to the Internet.

NAT: Problems with Firewalls

- What will happen if you want to run a server at home?
- The connections will start from the outside and there is no mapping in the NAT table.
- The packets will be dropped before reaching the server.
- Solution: Add static entries to the NAT table manually.

NAT: Static Entries



NAT Table

Src	Port	Newport
A	4563	7867
B	80	80 (manual)

1. B is running an HTTP server in the private network.
2. To allow connections from the Internet, the administrator adds a static entry in the NAT table to redirect any request to port 80 to B.

NAT: Problems with Firewalls.

- Some application eliminate this problem by having a proxy server forward the data.
- The computer in the private net makes a connection to the proxy server and then the proxy server forwards any data to the host in the private net.

NAT Configuration

- NAT boxes have a built in HTTP server that you can use to configure it or add static entries.
- For protection, any request to add a static entry in the NAT table or any configuration in the NAT has to come from a computer in the private Network.

Sockets and Ports

- A port defines an end-point (application) in the machine itself
- There are well-known ports:
  - HTTP Port 80
  - SSH Port 22
  - FTP Port 21
- If you are building an application that will be deployed globally, you may request your own port number.
- A socket is a file descriptor that can be used to receive incoming connections or to read/write data to a client or server.

Sockets and Ports

- A TCP connection is defined uniquely in the entire Internet by four values:

  <src-ip-addr, src-port, dest-ip-addr, dest-port>
- Example: A runs an HTTP server in port 80
- B connects to A's HTTP server using source port 5000
  - The connection is <IB, 5000, IA, 80>
- C connects also to A's HTTP server using src port 8000
  - The connection is <IC, 8000, IA, 80>
- Another browser in B using port 6000 connects to A
  - The connection is <IB, 6000, IA, 80>
- Another browser in C using port 5000 connects to A
  - The connection is <IC, 5000, IA, 80>
- The OS uses this 4 values to know what data corresponds to what application/socket.

Sockets API

- They were introduced by UNIX BSD (Berkeley Standard Distribution).
- They provide a standard API for TCP/IP.
- A program that uses sockets can be easily ported to other OS's that implement sockets: Example: Windows.
- Sockets were designed general enough to be used for other platforms besides TCP/IP. That also makes sockets more difficult to use.

Sockets API

- Sockets offer:
  - Stream interface for TCP.

    Read/Write is similar to writing to a file or pipe.
  - Message based interface for UDP

    Communication is done using messages.
- The first applications were written using sockets: FTP, mail, finger, DNS etc.
- Sockets are still used for applications where direct control of the network is required.
- Communication is programmed as a conversation between client and server mostly using ASCII Text.

Programming With Sockets

Client Side

```
 int cs =socket(PF_INET, SOCK_STREAM, proto)
…
Connect(cs, addr, sizeof(addr))
…
Write(cs, buf, len)
Read(cs, buf, len);
Close(cs)
```

See:

http://www.cs.purdue.edu/homes/cs354/lab5-http-server/client.cpp

Programming With Sockets

- Server Side

```
…
int masterSocket = socket(PF_INET, SOCK_STREAM, 0);
…
int err = setsockopt(masterSocket, SOL_SOCKET, SO_REUSEADDR, (char *) &optval, sizeof( int
      ) );
…
int error = bind( masterSocket, (struct sockaddr *)&serverIPAddress, sizeof(serverIPAddress) );
…
error = listen( masterSocket, QueueLength);
…
while ( 1 ) {
…
   int slaveSocket = accept( masterSocket,
      (struct sockaddr*)&clientIPAddress, (socklen_t*)&alen);
   read(slaveSocket, buf, len);
   write(slaveSocket, buf, len);
  close(slaveSocket);
}
```

- See: http://www.cs.purdue.edu/homes/cs354/lab5-http-server/lab5-src/daytime-server.cc

Client for Daytime Server

```
//------------------------------------------------------------------------
// Program:   client
//
// Purpose:   allocate a socket, connect to a server, and print all output
//
// Syntax:    client host port
//
//                 host  - name of a computer on which server is executing
//                 port  - protocol port number server is using
//
//------------------------------------------------------------------------

#define closesocket close
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
```

Client for Daytime Server

```
void
printUsage()
{
  printf( "Usage: client <host> <port> <name>\n");
  printf( "\n");
  printf( "          host: host where the server is running.\n");
  printf( "          port: port that the server is using.\n");
  printf( "\n");
  printf( "Examples:\n");
  printf( "\n");
  printf( "          client localhost 422422\n");
  printf( "          client lore 1345\n");
  printf( "\n");
}
```

Client for Daytime Server

```
int
main(int argc, char **argv)
{
  // Check command-line argument for protocol port and extract
  // host and port number
  if ( argc < 4 ) {
    printUsage();
    exit(1);
  }

  // Extract host name
  char * host = argv[1];

  // Extract port number
  int port = atoi(argv[2]);

  char * name = argv[3];

  // Initialize socket address structure
  struct  sockaddr_in socketAddress;

  // Clear sockaddr structure
  memset((char *)&socketAddress,0,sizeof(socketAddress));
```

Client for Daytime Server

```
// Set family to Internet
socketAddress.sin_family = AF_INET;

// Test for port legal value
if (port > 0) {
  socketAddress.sin_port = htons((u_short)port);
}
else {
  fprintf(stderr,"bad port number %s\n", argv[2]);
  exit(1);
}

// Get host table entry for this host
struct  hostent  *ptrh = gethostbyname(host);
if ( ptrh == NULL ) {
  fprintf(stderr, "invalid host: %s\n", host);
  perror("gethostbyname");
  exit(1);
}

// Copy the host ip address to socket address structure
memcpy(&socketAddress.sin_addr, ptrh->h_addr, ptrh->h_length);
```

Client for Daytime Server

```
// Get TCP transport protocol entry
  struct  protoent *ptrp = getprotobyname("tcp");
  if ( ptrp == NULL ) {
    fprintf(stderr, "cannot map \"tcp\" to protocol number");
    perror("getprotobyname");
    exit(1);
  }

  // Create a tcp socket
  int sock = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
  if (sock < 0) {
    fprintf(stderr, "socket creation failed\n");
    perror("socket");
    exit(1);
  }
```

Client for Daytime Server

```
// Connect the socket to the specified server
if (connect(sock, (struct sockaddr *)&socketAddress,
        sizeof(socketAddress)) < 0) {
  fprintf(stderr,"connect failed\n");
  perror("connect");
  exit(1);
}

// In this application we don't need to send anything.
// For your HTTP client you will need to send the request
// as specified in the handout using send().

int m = read(sock,buffer,100);
buffer[m]=0;
printf("buffer=%s\n", buffer);

write(sock, name, strlen(name));
write(sock,"\r\n",2);
```

Client for Daytime Server

```
// Receive reply
  // Data received
  char buf[1000];
  int n = recv(sock, buf, sizeof(buf), 0);
  while (n > 0) {
    // Write n characters to stdout
    write(1,buf,n);

    // Continue receiving more
    n = recv(sock, buf, sizeof(buf), 0);
  }
```

Client for Daytime Server

```
// Close the socket.
  closesocket(sock);

  // Terminate the client program gracefully.
  exit(0);
}
```

Daytime Server

```
const char * usage =
"                                                               \n"
"daytime-server:                                                \n"
"                                                               \n"
"Simple server program that shows how to use socket calls       \n"
"in the server side.                                            \n"
"                                                               \n"
"To use it in one window type:                                  \n"
"                                                               \n"
"   daytime-server <port>                                       \n"
"                                                               \n"
"Where 1024 < port < 65536.             \n"
"                                                               \n"
"In another window type:                                       \n"
"                                                               \n"
"   telnet <host> <port>                                        \n"
"                                                               \n"
"where <host> is the name of the machine where daytime-server  \n"
"is running. <port> is the port number you used when you run   \n"
"daytime-server.                                               \n"
"                                                               \n"
"Then type your name and return. You will get a greeting and   \n"
"the time of the day.                                          \n"
"                                                               \n";
```

Daytime Server

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
```

Daytime Server

```
int QueueLength = 5;

// Processes time request
void processTimeRequest( int socket );
// Get time of day
  time_t now;
  time(&now);
  char  *timeString = ctime(&now);

  // Send name and greetings
  const char * hi = "\nHi ";
  const char * timeIs = " the time is:\n";
  write( fd, hi, strlen( hi ) );
  write( fd, name, strlen( name ) );
  write( fd, timeIs, strlen( timeIs ) );

  // Send the time of day
  write(fd, timeString, strlen(timeString));

  // Send last newline
  const char * newline="\n";
  write(fd, newline, strlen(newline));
}
```

Daytime Server

```
int
main( int argc, char ** argv )
{
  // Print usage if not enough arguments
  if ( argc < 2 ) {
    fprintf( stderr, "%s", usage );
    exit( -1 );
  }

  // Get the port from the arguments
  int port = atoi( argv[1] );

  // Set the IP address and port for this server
  struct sockaddr_in serverIPAddress;
  memset( &serverIPAddress, 0, sizeof(serverIPAddress) );
  serverIPAddress.sin_family = AF_INET;
  serverIPAddress.sin_addr.s_addr = INADDR_ANY;
  serverIPAddress.sin_port = htons((u_short) port);
```

Daytime Server

```
// Allocate a socket
 int masterSocket =  socket(PF_INET, SOCK_STREAM, 0);
 if ( masterSocket < 0) {
   perror("socket");
   exit( -1 );
 }

 // Set socket options to reuse port. Otherwise we will
 // have to wait about 2 minutes before reusing the sae port number
 int optval = 1;
 int err = setsockopt(masterSocket, SOL_SOCKET, SO_REUSEADDR,
              (char *) &optval, sizeof( int ) );

 // Bind the socket to the IP address and port
 int error = bind( masterSocket,
           (struct sockaddr *)&serverIPAddress,
           sizeof(serverIPAddress) );
 if ( error ) {
   perror("bind");
   exit( -1 );
 }
```

Daytime Server

```
// Put socket in listening mode and set the
 // size of the queue of unprocessed connections
 error = listen( masterSocket, QueueLength);
 if ( error ) {
   perror("listen");
   exit( -1 );
 }

 while ( 1 ) {

   // Accept incoming connections
   struct sockaddr_in clientIPAddress;
   int alen = sizeof( clientIPAddress );
   int slaveSocket = accept( masterSocket,
                 (struct sockaddr *)&clientIPAddress,
                 (socklen_t*)&alen);
```

Daytime Server

```
if ( slaveSocket < 0 ) {
      perror( "accept" );
      exit( -1 );
    }

    // Process request.
    processTimeRequest( slaveSocket );

    // Close socket
    close( slaveSocket );
  }

}
```

Daytime Server

```
void
processTimeRequest( int fd )
{
  // Buffer used to store the name received from the client
  const int MaxName = 1024;
  char name[ MaxName + 1 ];
  int nameLength = 0;
  int n;

  // Send prompt
  const char * prompt = "\nType your name:";
  write( fd, prompt, strlen( prompt ) );

  // Currently character read
  unsigned char newChar;

  // Last character read
  unsigned char lastChar = 0;
```

Daytime Server

```
//
// The client should send <name><cr><lf>
// Read the name of the client character by character until a
// <CR><LF> is found.
//

while ( nameLength < MaxName &&
    ( n = read( fd, &newChar, sizeof(newChar) ) ) > 0 ) {

  if ( lastChar == '\015' && newChar == '\012' ) {
    // Discard previous <CR> from name
    nameLength--;
    break;
  }

  name[ nameLength ] = newChar;
  nameLength++;

  lastChar = newChar;
}

// Add null character at the end of the string
name[ nameLength ] = 0;

printf( "name=%s\n", name );
```

Daytime Server

```
 // Get time of day
  time_t now;
  time(&now);
  char  *timeString = ctime(&now);

  // Send name and greetings
  const char * hi = "\nHi ";
  const char * timeIs = " the time is:\n";
  write( fd, hi, strlen( hi ) );
  write( fd, name, strlen( name ) );
  write( fd, timeIs, strlen( timeIs ) );

  // Send the time of day
  write(fd, timeString, strlen(timeString));

  // Send last newline
  const char * newline="\n";
  write(fd, newline, strlen(newline));
}
```

Types of Server Concurrency

- Iterative Server
- Fork Process After Request
- Create New Thread After Request
- Pool of Threads
- Pool of Processes

Iterative Server

```
void iterativeServer( int masterSocket) {
   while (1) {
      int slaveSocket =accept(masterSocket,
              &sockInfo, &alen);
      if (slaveSocket >= 0) {
       dispatchHTTP(slaveSocket);
      }
   }
}
```

```
Note: We assume that dispatchHTTP itself
   closes slaveSocket.
```

Fork Process After Request

```
void forkServer( int masterSocket) {
  while (1) {
      int slaveSocket = accept(masterSocket,
                         &sockInfo, &alen);
     if (slaveSocket >= 0) {
       int ret = fork();
`      if (ret == 0) {
            dispatchHTTP(slaveSocket);
            exit(0);
       }
       close(slaveSocket);
     }
  }
}
```

Create Thread After Request

```
void createThreadForEachRequest(int masterSocket)
{
  while (1) {
    int slaveSocket = accept(masterSocket, &sockInfo, &alen);
    if (slaveSocket >= 0) {
      // When the thread ends resources are recycled
      pthread_attr_t attr;
      pthread_attr_init(&attr);
      pthread_attr_setdetachstate(&attr,
                     PTHREAD_CREATE_DETACHED);
      pthread_create(&thread, &attr,
              dispatchHTTP, (void *) slaveSocket);
    }
  }
}
```

Pool of Threads

```
void poolOfThreads( int masterSocket ) {
   for (int i=0; i<4; i++) {
    pthread_create(&thread[i], NULL, loopthread,
                   masterSocket);
   }
   loopthread (masterSocket);
}

void *loopthread (int masterSocket) {
  while (1) {
    int slaveSocket = accept(masterSocket,
                      &sockInfo, &alen);
    if (slaveSocket >= 0) {
      dispatchHTTP(slaveSocket);
    }
  }
}
```

Pool of Processes

```
void poolOfProcesses( int masterSocket ) {
   for (int i=0; i<4; i++) {
     int pid = fork();
     if (pid ==0) {
       loopthread (masterSocket);
     }
   }
   loopthread (masterSocket);
}

void *loopthread (int masterSocket) {
  while (1) {
    int slaveSocket = accept(masterSocket,
                     &sockInfo, &alen);
    if (slaveSocket >= 0) {
      dispatchHTTP(slaveSocket);
    }
  }
}
```

Notes:

- In Pool of Threads and Pool of processes, sometimes the OS does not allow multiple threads/processes to call accept() on the same masterSocket.
- In other cases it allows it but with some overhead.
- To get around it, you can add a mutex_lock/mutex_unlock around the accept call.

```
mutex_lock(&mutex);
int slaveSocket = accept(masterSocket,
                         &sockInfo, 0);
mutex_unlock(&mutex);
```

- In the pool of processes, the mutex will have to be created in shared memory.

SQL and Relational Databases

Relational Databases

- You have learned how to store data persistently into files.
- However, when using plane files, you have to write functions to parse the information, to search it etc.
- Relational databases (or just called Databases) store the information in tables.
- A table is made of many rows and columns, where one row stores a record and a column stores an attribute of the record.
- Databases provides functions to add, delete, and search records in a table.
- SQL (Structured Query Language) is the standard language used to manipulate databases.

Available SQL databases

- MySQL
  - http://www.mysql.com
  - Very popular free database
- Oracle
  - http://www.oracle.com
  - Very popular database for enterprise use
- Microsoft SQL
  - http://www.microsoft.com/sqlserver
  - Also very popular.

Structured Query Language (SQL)

- All major languages let you communicate with a database using SQL.
- Databases have a GUI program that lets you manipulate the database directly and can be used for database administration.
- You can think of a database as a group of named tables with rows and columns.
- Each column has a name and each row contains a set of related data.

Single Table

Title	ISBN	FName	LName	Price	Publisher	URL
A Guide to the SQL Standard	0-201-96426-0	Alexander	Christopher	47.95	Addison-Wesley	www.aw-bc.com
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	Frederick P.	Brooks	65.00	John Wiley & Sons	www.wiley.com
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	Smith	Peter	65.00	John Wiley & Sons	www.wiley.com

Structuring your information in Tables

- As part of the design of a project, you need to design the tables in a database.
- You may store your information in multiple tables to reduce the amount of repeated information.
- In the example before we could store, all the information in a single table but that would lead to many repeated entries.

SQL by Example (from textbook)

Authors Table

Autor_ID	Name	Fname
ALEX	Alexander	Christopher
BROO	Brooks	Frederick P.
SMITH	Smith	Peter

Books Table

Title	ISBN	Publisher_ID	Price
A Guide to the SQL Standard	0-201-96426-0	0201	47.95
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	0407	65.00

SQL by Example (from textbook)

BookAuthors Table

ISBN	Author_ID	Seq_No
0-201-96426-0	ALEX	1
0-201-96426-0	BROO	2
0-19-501919-9	DAR	1

Publishers Table

Publisher_ID	Name	URL
0201	Addison-Wesley	www.aw-bc.com
0407	John Wiley & Sons	www.wiley.com

Primary Keys

- One or more of the columns of the table are defined as “Primary Keys”.
- The value in each column of a primary key is recommended to be unique.
- A separate data structure called “B-tree” is created for this “Primary Key” for quick access.
- Searches on this binary key will be at least O(log n), compared to searches in other columns that will be sequential.

SQL by Example

- By convention SQL keywords are written in uppercase.
- SELECT * FROM Books
  - This query returns all rows in the Books table.
  - SQL statements always require FROM
- SELECT ISBN, Price, Title
  FROM Books
  - This query returns a table with only the ISBN, price and title columns from the Books table.

SQL by Example

```
SELECT ISBN, Price, Title
   FROM Books
   WHERE Price <=29.95
```

- This query returns a table with the ISBN, Price and Title from the Books table but only for the books where the price is less or equal to 29.95.

SQL by Example

SELECT ISBN, Price, Title

FROM Books

WHERE Title NOT LIKE “%n_x%”

Returns a table with ISBN, Price and Title as columns excluding the books that contain Linux or UNIX in the title.

The “%” character means any zero or more characters. “_” means any single character.

SQL by Example

```
SELECT  Title, Name, URL
   FROM Books, Publishers
   WHERE Books.Publisher_ID=Publishers.Publisher_ID
```

It returns a table with the Title, Name of publisher, and URL from Books and Publishers.

Title	Name	URL
A Guide to the SQL Standard	Addison-Wesley	www.aw-bc.com
A Pattern Language: Towns, Buildings, Construction	John Wiley & Sons	www.wiley.com

SQL by Example

You can also use SQL to change data inside a database.

```
UPDATE Books
  SET Price = Price – 5.00
  WHERE Title Like “%C++%”
```

This reduces the price by $5.00 for all books that have C++ in the title.

SQL by Example (from textbook)

You can also delete rows with SQL

```
DELETE FROM Books
   WHERE Title Like “%C++%”
```

This deletes all books that have C++ in the title.

SQL by Example

Use INSERT to insert a new row in the table.

INSERT INTO Books

VALUES (‘A Guide to the SQL Standard’, ‘0-201-96426-0’, ‘0201’, 47.95)

This inserts a new book in the Books table.

SQL by Example

You can also create a new table using SQL

```
CREATE TABLE Books
   (
      TITLE CHAR(60),
      ISBN CHAR(13),
      Publisher_ID CHAR(6),
      Price DECIMAL(10,2)
)
```

SQL Tutorials

For more information about SQL, see the SQL tutorial in

http://www.w3schools.com/sql/default.asp

You can also run some SQL examples there.

Running SQL in the SSLAB machines

- The explanation of how to run mysql inside your account in the sslab machines is in :
  - http://support.cs.purdue.edu/help/MySQL_mini-HOWTO,_Linux
- The instructions allow you to run the database in the background but it will allow only to connect clients in the same machine.
- Here is the mysql tutorial. Follow the tutorial to get familiar with mysql.
  - http://dev.mysql.com/doc/refman/5.1/en/tutorial.html

Running mysql in the sslab machines

To run mysql type:

```
bash> mysql -u root
mysql> CREATE DATABASE menagerie;
mysql> USE menagerie
mysql> SHOW TABLES;
mysql> CREATE TABLE pet (name VARCHAR(20), owner VARCHAR(20),
    -> species VARCHAR(20), sex CHAR(1), birth DATE, death DATE);
mysql> SHOW TABLES;
mysql> DESCRIBE pet;
mysql> INSERT INTO pet
    -> VALUES ('Puffball','Diane','hamster','f','1999-03-30',NULL);
mysql> select * from pet;
```

Software Development in Teams

Joel's Test

- Joel Spolsky is a software developper who has written several books on Software Engineering.
- His blog is in http://www.joelonsoftware.com
- He is a proponent of "Extreme Programming" a Software Engineering Technique for software development.
- This is "Joel's Test" that summarizes what is important in practical Software Engineering.
- See http://www.joelonsoftware.com/articles/fog0000000043.html

Joel Test

The Joel Test

1. Do you use source control?
2. Can you make a build in one step?
3. Do you make daily builds?
4. Do you have a bug database?
5. Do you fix bugs before writing new code?
6. Do you have an up-to-date schedule?
7. Do you have a spec?
8. Do programmers have quiet working conditions?
9. Do you use the best tools money can buy?
10. Do you have testers?
11. Do new candidates write code during their interview?
12. Do you do hallway usability testing?

Total = 100* (#Yes/12)

What is XP Programming?

- XP stands for “Extreme Programming”
- It is a practical methodology for software development.
- It gives a list of rules that have been proven successful in software development.
- Instead of emphasizing an straight “design-and-program” approach, it encourages an iterative approach.

When to use XP Programming?

- It is for projects that have some risk of completion
  - Design changes continuously.
  - Customer does not have a clear idea of what has to be accomplished.
- Used for small groups (2-12 people)
- Requires good communication between customers and programmers.

Rules and Practices of XP

- Planning
  - User Stories
  - Release Planning
  - Frequent Small Releases
  - Measure Project Development
  - Iterative Development
  - Move People Around
  - Daily Standup Meeting
  - Fix XP when it breaks
- Coding
  - Have customer available
  - Follow coding standards
  - Code unit-test first
  - Use pair programming
  - One code integration at a time
  - Integrate often
  - Collective ownership
  - Leave optimization until the end.
  - No overtime

Rules and Practices in XP (cont.)

- Designing
  - Keep it Simple
  - Choose System Metaphor
  - Use CRC cards
  - Create “Spike Solutions”
  - Do not add functionality early
  - Refactor
- Testing
  - All code must have uni tests
  - All code must pass all uni tests before integration
  - When a bug is found create a test.
  - Create acceptance tests
- See “Extreme Programming: A gentle introduction”; http://www.extremeprogramming.org/

Internal Development Web Site

- Your development team should have an internal web page with links to:
  - Sources
  - How to build the system
  - Results of last daily build
  - Bug Tracking system
  - Design documents
  - Directory of members

Internal Development Web Site

- Do not leave the knowledge in the head of the programmers.
- Everything considered important for the project should be written.
- Some people use wiki’s that are easy to modify.
- A wiki is a web system that allows adding information using the browser.
- The wiki should be just part of the internal web site but not the whole internal web site.
- It is easier to add some directories to a htdocs directory in the web server than attach it to a wiki, specially if the documents are already in HTML.

Source Control

- Source Control will allow multiple developers to modify the same source code.
- It will keep track of the changes. You can always go back to any point in time.
- It also makes merges easier. If two programmers modify different sources or different sections of the code, it will automatically merge them.
- If two programmers modify the same section of the code, it will tell that there is a conflict and it will start a three-way merge program: (Parent’s, Other’s, Yours)

Source Control Programs

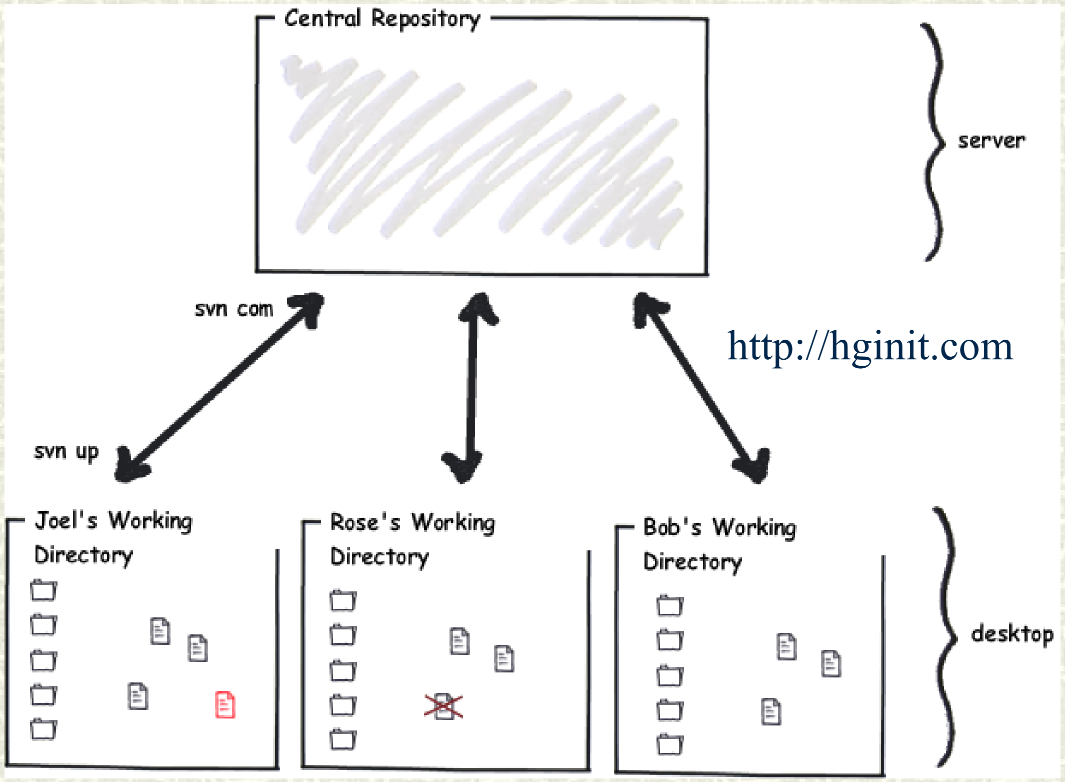
- CVS –
  - The oldest of all source controls.
- SVN –
  - Rewrite of CVS.
  - Centralized repository. Very popular.
- Perforce
  - Also very popular
  - Not Free
- Mercurial –
  - Distributed Source Control.
  - No need for one centralized server.
  - Uses a local repository.
- GIT
  - Also distributed source control
  - Fast.
  - Written by Linus Torvalds for the Linux Kernel
- Any of these source control programs will be fine as long as you use it correctly and get the best of it.

Uses of Source Control

- Keep Track of Changes of multiple programmers.
- Makes merges of multiple programmers easier.
- You can go back in time.
- You can query who modified a file and when.
- You can learn what files need to be changed to implement a feature.
- You can find out what changes broke the daily build.
- You can evaluate at what level people contributed to a project.
- You can peer review changes
- You have a backup of your sources

Centralized Source Control

- Centralized Source Control systems like SVN have a central repository.



http://hginit.com

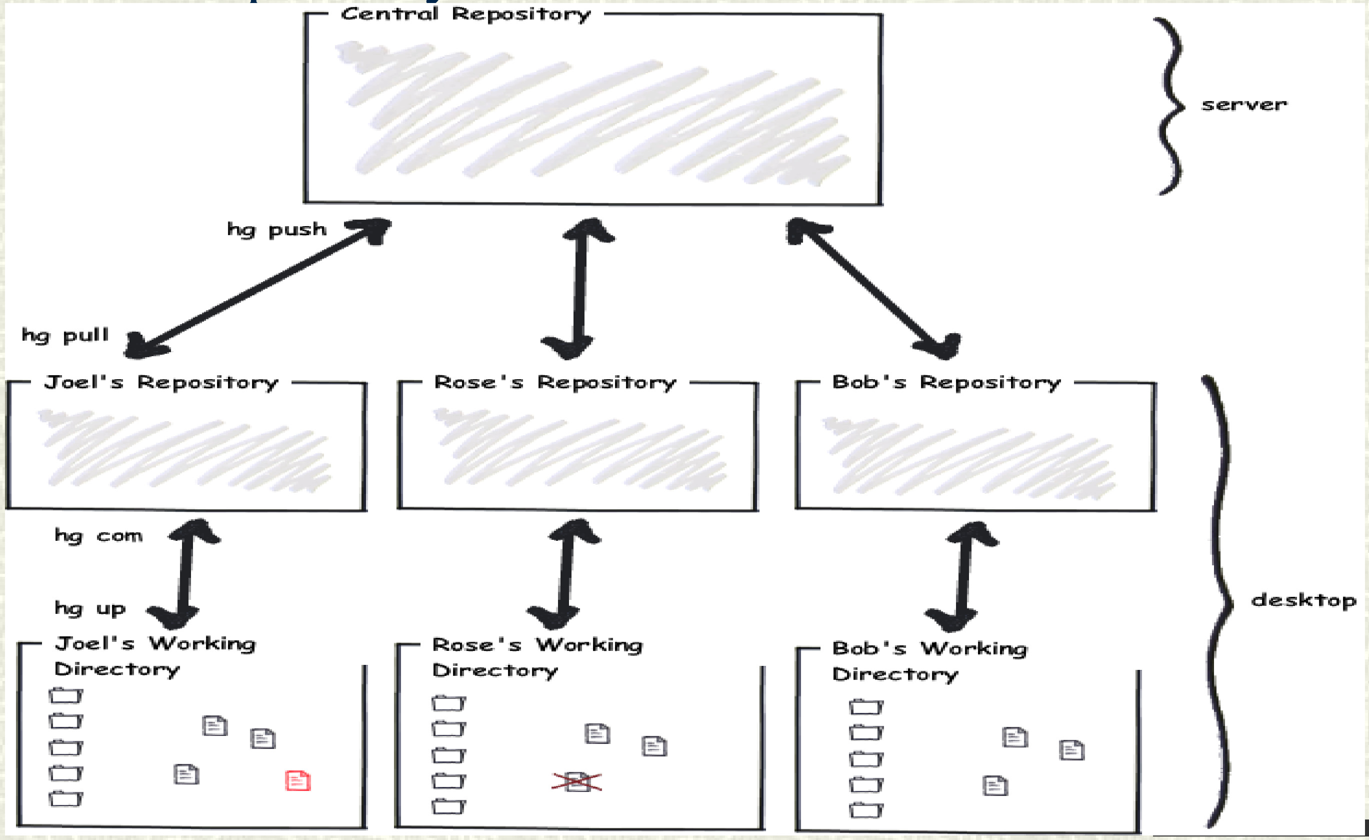
- Commands:
  - svn commit
  - svn update

Distributed Source Control

- Distributes Source Control Systems do not have a centralized repository.
- Instead, the programmers will have a local repository.
- Programmers can exchange changes without the need of a centralized repository.
- Most of the time users of a distributed source control systems have a common place with the most recent version.

Distributed Source Control

- Distributed Source Control Systems like Mercurial (HG) add a local repository.



Centralized vs. Distributed Source Control

- Both work
- Distributed Source Control for programmers that have to work a very long time on the sources without submitting to the common code.
- They still continue to update the local repository. The list of changes help when "pushing" the changes to the common repository.
- The list of code changes can be send by e-mail.

Testing

- The most practical way to improve the quality of a program is through testing.
- Tests have to run automatically if possible.
- Manual testing is also useful but it is costly in human hours.

Who writes tests?

- The programmer
  - The programmer has to write tests to verify that the software works as expected.
  - The tests may have knowledge about the internals of the product.
  - Put tests in source control.
  - Treat your tests as part of the source code since they have the same importance.
- Quality Assurance Department (QA)
  - It is a team independent of the development team.
  - They write tests that do not have any knowledge of the internals. They consider product as a black-box.
  - It is important that people independent from the team test the product.
  - They should test the product at least at pre-release.

What if there is no QA?

- If your organization do not have money for a QA department, before each version ask the development team to work as QA testers.
- A developer should not be assigned to test features he/she implemented.

Types of Tests

- Unitests
  - A group of tests that test a specific class.
  - Every class should have unitests.
  - Written by programmers
- System Tests
  - Test a specific subsystem of the product. Test multiple classes involved in a specific feature.
  - Written by QA and programmers

Types of Tests

- Regression Tests
  - Test written to reproduce a bug and then it is used to verify that the bug is fixed.
  - If no regression test is written, there is a risk that the bug will be reintroduced.
  - Written by programmers and QA.
- Acceptance Tests
  - Evaluate the quality of the software before a release.
  - The tests tell if the product is ready for prime time or not.
  - Written by QA.

When to Run Tests

- After writing code.
- Before committing a change to source control so other people will get good sources when updating.
- Every day during the daily build to verify that the main repository is not broken.
- Some systems trigger an automatic build when somebody commits code into source control.
- The easier it is to run the tests, the more often they will be run and the higher the quality of your code.
- Do your best to not break the daily build but that should not prevent you from implementing new features.
- The success of a software project is measured by both the features and quality of code.

Tracking Bugs and Features

- It is important to have a data base with the existing bugs.
- Features can be tracked the same way as bug and often they use the same database.
- The bug/feature cycle is:
  - Create a bug/feature report.
  - The Product Manager assigns a priority and severity.
    - Priority – Importance for the organization.
      - 1 –Finish it as soon as possible.
      - 2 – Make sure that it is in next release
      - 3 – Try to put it in next release.
    - Severity – How it impacts the user.
      - 1 – User absolutely cannot use the product without fixing this bug.
      - 2 – After a workaround the user can use product.
      - 3 – User can use product but bug makes use somehow difficult.

Tracking Bugs and Features

- The bug/feature cycle (cont.)
  - Assign the bug to a programmer.
  - The programmer analyzes the bug an puts an estimate of time.
  - The programmer fixes the bug and creates a regression test.
  - The programmer submits the fix to code review.
  - Once accepted, the fix is committed to source control and the bug report is passed to QA to verify that the bug has been fixed.
  - The bug report is closed. Closed bugs are reported in the release notes.

Bug Tracking Programs

- There are commercial and Open Source software for buck tracking:
  - Bugzilla – Part of the mozilla project.
  - Phabricator - Bug tracking open source. Used in Facebook.
- See bugzilla in action:
  - https://bugzilla.mozilla.org/
- Also, browse the changes submitted to mozilla source control. Select one of the repositories.
  - http://hg.mozilla.org/

Refactoring

- Very seldom you will start code from scratch.
- More often you will add or maintain to existing code.
- For this reason you have to write your code to be maintainable, so you or other people can understand it.
- After implementing a feature or fixing a bug, see opportunities to make your code look better.
- See http://wiki.java.net/bin/view/People/SmellsToRefactorings

Refactoring

- Comments
  - Should only be used to clarify "why" not "what".
  - Comments are important.
  - Use Javadocs or other ways to combine comments and documentation.
- Long Method
  - The longer the method the harder it is to see what it is doing.
  - Extract method.
- Long Parameter List
  - Don't pass in everything the method needs; pass in enough so that the method can get to everything it needs.
  - Use a parameter object that contains multiple parameters

Refactoring

- Duplicated Code
  - Extract Method
- Large Class
  - A class that is trying to do too much can usually be identified by looking at how many instance variables it has. When a class has too many instance variables, duplicated code cannot be far behind.
  - Extract Class
    Extract Subclass
- Type Embedded in Name
  - Avoid redundancy in naming. Prefer schedule.add(course) to schedule.addCourse(course)
  - Rename Method
- Uncommunicative Name
  - Choose names that communicate intent (pick the best name for the time, change it later if necessary).
  - Rename Method

Refactoring

- Inconsistent Names
  - Use names consistently.
  - Rename Method
- Dead Code
  - A variable, parameter, method, code fragment, class, etc is not used anywhere (perhaps other than in tests).
  - Delete the code.
- Speculative Generality
  - Don't over-generalize your code in an attempt to predict future needs.
  - If you have abstract classes that aren't doing much use Collapse Hierarchy
    Remove unnecessary delegation with Inline Class
    Methods with unused parameters - Remove Parameter
    Methods named with odd abstract names should be brought down to earth with Rename Method

Introduction to Software Patterns

- They are reusable design solutions to reoccurring problems.
- Modeled after Architectural design
- Patterns have been called the next best software improvement since object-oriented programming.
- Introduced in the book “Design Patterns: Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson and Vlissides [GoF95]

Introduction to Patterns

- Modeled after the books: “A pattern language: Towns, Buildings, Construction” by architect Christopher Alexander.
- Examples:
  - Symmetry is good
  - Use half-round arches to support bridges and in doors.
- Same equivalent ideas exist in software design

Parts of a Software Pattern

- Pattern Name
- Synopsis – One to two sentence description.
- Context – Describes the problem the pattern addresses.
- Forces – What leads to the solution proposed.
- Solution – Describes the general solution.
- Consequences – Implications good and bad.
- Implementation – What to consider when programming the solution
- Related Patterns

Proxy Pattern

- Synopsis: Forces method calls to an object to occur through a proxy object.
- Context:
  - A proxy may give the illusion that the object is local when it is in a different machine (remote-proxy).
  - A proxy controls access (access proxy).
  - Proxy acts as a logger object.
  - Proxy may act as a cache.
- Forces:
  - We want to add functionality that is orthogonal to the object itself.

Proxy

- Solution: Both proxy and service-providing object can inherit from a common class or implement a common interface.
- Consequences: Service-providing object is used transparent to the object and to the users of the object.

Proxy

- Implementation/Code Example:

```
interface Table {
void add( string key, void * item);
}
class MyTable: implements Table {
void add( string key, void * item);
}
class TableLogger implements Table {
Table mytable;
TableProxy( Table t ) { mytable = t; }
void add (string key, void * item) {
        log( “add”, key, item );
        mytable.add( key, item );
}
}
```

Command [GoF95]

- Synopsis:
  - Encapsulate commands in objects so you can control selection and sequencing, queue them, undo them and manipulate them.
- Context:
  - Imagine you want to design a graphic editor and you want to undo/redo commands.
  - You may have each command be an object that provides a do() and undo() methods.
  - The commands will be stored in a queue. To undo one command execute undo() in last command. Alternatively clean and do all the commands except last one.

Command [GoF95]

- Forces:
  - We need undo/redo management.
  - Sequence of commands can be stored for future testing.
- Solution:
  - Create abstract interface with do() and undo() methods. Have command objects implement abstract interface. Have a command manager object store commands in queue and performed do/redo operations.

Command [GoF95]

- Implementation:

```
public abstract class Command {
  public abstract boolean doIt();
  public abstract boolean undoIt();
}
public abstract class CommandManager {
   public void invokeCommand( Command c) {
      if ( c instanceof Undo) undoLast();
      else if ( c instanceof Do) redoLast();
     else { c.doIt(); addToHistory( c);
     }
}
```

Command [GoF95]

- Consequences:
  - Flexibility in sequencing of commands.
  - Allow to replay commands at any time.

Observer [GoF95]

- Synopsis:
  - Allow to dynamically register dependencies between objects, so that an object will notify those objects that are dependent on it when its state changes.
- Context:
  - You have a directory viewing program that shows the contents of the directory. The program registers a “Listener” in the directory object to be notified of changes in the directory.

Observer [GoF95]

- Forces:
  - An instance of one class needs to notify the instance of another class that the state changed.
  - You may have a one-to-many dependency that may require one object top notify multiple objects.
- Solution:
  - Implement a Listener abstract class. The observer object implements this interface and registers itself as a listener in the observable object.
  - The observer object is notified when the state changes.

Observer [GoF95]

- Implementation

```
interface DirChangeListener {
    void contentChanged(Directory d)
}
Class DirectoryBrowser
    implements DirChangeListener {
  . . .
}
Class Directory {
    public registerListener( DirChangeListener l );
…
}
```

- Consequences:
  - Cyclic dependencies may arise

Visitor [GoF 95]

- Synopsis:
  - When you need to do operations in a complex structure, you can separate the operation from the structure by using a visitor class.
- Context:
  - Suppose you want the ability of creating a table of contents in a word processor.
  - The word processor may allow a way to iterate over all the paragraphs and document structures by calling a visitor object.
  - You may write a visitor object that extracts the titles necessary for the table of contents.

Visitor [GoF95]

- Forces:
  - There are a variety of operations that need to be performed to an object structure.
  - The object structure is composed of objects that belong to different classes.
- Solution:
  - Implement an abstract class Visitor that he concrete visitor implements and that has multiple visit methods for different types of objects.
  - The ObjectStructure implements a visit method that takes a Visitor object as argument.
  - The methods in theVisitor object is called by the visit method of the ObjectStructure for every object in the

Visitor [GoF95]

- Implementation

```
Interface WordProcessorVisitor {
   void visit(Paragraph p);
   void visit(Title t);
}
public class WordProcessor {
   ….
   public visit(WordProcessorVisitor v) {…}
}
Public class CreateTOC implements WordProcessorVisitor {
   void visit(Paragraph p) {…}
   void visit(Title t) {…}
}
```

Bibliography of Software Patterns

- [GoF95]Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. “Design Patterns: Elements of Reusable Object-Oriented Software”. Reading Mass. Addison Wesley, 1995.
- [Grand98] Mark Grand. “Patterns in Java. A Catalog of Reusable Design Patterns in Java Illustrated with UML.” Wiley 1998.

Program Optimization

When Writing a Program

1. Keep It Simple
   - Write the simplest implementation that provides the functionality you need.
2. Test
   - Test the Program to make sure that it does what it is supposed to do.
3. Refactor
   - Make your code simple and readable so other people can easily understand it and maintin it.
4. Optimize program only if necessary
   - Only if your code does not run fast enough, profile it with an execution profiling tool to find where it spends most of the time.
   - You will be surprised that maybe about 0.05% of your code needs to be optimized.

Premature Optimization

- Programmers often optimize in advance parts of the code that do not need to be optimized.
- Only small portions of code need to be optimized.
- You will be able to leave most of the code intact, well written, easy to read.
- This is good because optimizing code may make the code difficult to understand.

Profiling Tools

- Code Instrumentation
  - Add instructions into the code before a method starts and after a method returns to measure the execution of a method.
- Statistical Sampling
  - Sample the program at regular intervals to find out what the program is doing.

Profiling with Code Instrumentation

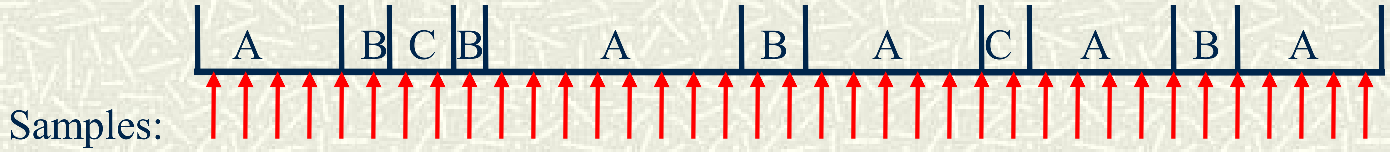
- Insert code to measure how long methods take.
- Intrusive but effective.
- Sometimes the inserted code adds overhead to the execution.
- Execution Profiler Programs with Code Instrumentation:
  - VTune (For x86)
  - JProfiler (For Java)
  - Many others

Statistical Profiling

- At regular intervals it samples what the program is doing.
- It usually happens during context switch with the timer interrupt so there is little overhead.
- The context switch handler records the Program Counter at the time of the context switch.
- The profiler creates a histogram of the number of samples and the function executed in each sample.
- The function with the most samples is the one that is consuming most of the CPU time.
- Execution Profiler Programs with Statistical Profiling:
    - Prof and GProf (Traditional UNIX Profiler)

Statistical Profiling

```
A() {
…
}

B() {
…
}

C() {
…
}
```



T=0.01Secs

A = 25 samples x .01secs/sample = .25secs

B = 8 samples x .01secs/sample = .08secs

C = 4 samples x .01secs/sample = .04 secs

Final Review

- Internet
  - Internet Architecture: Routers, Hosts, Networks
  - Internet Layering
  - IP Addressing
  - Routing
  - ARP – Address Resolution Protocol
  - DNS – Domain Name Service
  - DHCP – Dynamic Host Configuration Protocol
  - UDP – User Datagram Protocol

Final Review

- TCP
  - Adaptive Retransmission
  - Cumulative Acknowledgements
  - Fast Retransmission
  - Flow Control
  - Congestion Control
  - Reliable Connection and Shutdown
- NAT – Network Address Translation.
  - How it works.
  - Problems with Firewalls.

Final Review

- Sockets
  - Ports
  - Client API
  - Server API
- Server Concurrency
  - Iterative Server
  - Fork Process After a Request
  - New Thread After Request
  - Pool of Threads
  - Pool of Processes

Final Review

- Software Engineering
  - Joel’s Test
  - XP Programming
    - What is XP Programming
    - Rules and Practices of XP
      - Planning
      - Coding
      - Designing
      - Testing

Final Review

- Internal Development Website
- Source Control
  - Centralized and Distributed
  - Uses of Source Control
- Testing
  - Why Testing
  - Who writes tests
  - Types of Tests
    - Unitests
    - System Tests
    - Regression Tests
    - Acceptance Tests
  - When to Run Tests

Final Review

    - Tracking Bugs and Features
    - Refactoring
- Introduction to Software Patterns
    - Parts of a Software Pattern
    - Proxy Pattern
    - Command Pattern
    - Observer Pattern
    - Visitor Pattern
- Execution Profiling
    - Premature Optimization
    - Code Instrumentation Profiling
    - Statistical Profiling

Final Review

- SQL
  - What is SQL
  - SQL Commands and Examples