Process Address Space

What is operating system?

- Operating system is a software that provides an easy to use interface between user and machine.
- Example: Copying one file to another
- Using OS requires a simple command or right click and select “copy and paste” option
- Without OS: an user needs to
  - Read data from hard disk blocks (first file A) into memory
  - Write data to other disk blocks for new file B
  - Needs to update directory structure so that blocks related to other files are not overwritten

OS Functions

- There are three main functions that an OS performs
  - Provides an easy-to-use interface between user and machine. User need not understand functioning of hardware. Irrespective of underlying hardware, interface to user remains the same.
  - Manages computer resources on behalf of user. Resources are hardware (CPU/Memory/IO device/Netowrk interface cards/GPGPU/video RAM/priner/hard disk/removable media etc.) as well as software (processes/files/application program etc.)
  - Security: ensures isolation between processes and data of one user from other users. Also protects system programs as well as data on the system from external attackers.

OS: Views

- Extended machine
- Virtual Machine
- Resource manager
  - Multiplexes resources for efficiency

Evolution of OS

- Single user systems
- OS – loader + library of common routines
- Called supervisor or control program
  - Loads program and library functions from tape to memory
  - Transfers control to program
  - After program execution, copies CPU register values to core memory (core dumped). In case of a n error, a programmer shall look at the dump to analyze what went wrong

Batch operating systems

- All programs written in one language to be run as a single batch
- Shall reduce time for loading library tapes – done manually
- OS = sequencer + loader + output program

Multiprogramming

- Running multiple programs concurrently
  - Keeping many programs in memory,
  - running one of them (say program A)and
  - switch to another (say program B) when I/O required for A
- Aim was to improve resource utilization
  - Idle time of CPU should be minimized
- Programs could belong to different users - multi-user
- A multiuser OS is multiprogramming but not vice-versa.

Process Address Space

- How much memory is needed for an executable program?
  - Memory to be allocated for machine code
  - Memory to be allocated for all variables, constants and labels referred to in code
  - Memory required for code linked from libraries
  - Memory needed for maintaining
    - **Stack (to facilitate function calls)**
    - **Heap (memory allocation at run time)**
- Requires program to be translated from high level language to machine language

Program Translation

- Compiler
  - Translates HLL program into machine equivalent
  - Creates table of incomplete instructions for unresolved addresses
    - **Library functions, Extern variables**
- Linker
  - Static linking: Concatenates library with program and resolves call to library functions
  - Dynamic linking: Address is resolved at runtime; OS maintains a table of DLL files loaded in the memory
- Loader

Program

- Program: a collection of machine instructions; each instruction is of format opcode [operand1, [operand2]]
- Identifier: needs to be mapped to an address
  - Variable, constant, function name, label
- Statements:
  - Data declaration: type of data (used by compiler to assess bytes required to store data)
  - Assignment
  - Expressions
  - Control constructs
  - Function calls

Statements

- X = Y
  - MOV accumulator, Y
  - MOV X, accumulator
- Expression: A + B * C
  - MOV accumulator, B
  - MULT accumulator, C
  - ADD accumulator, A

Compiler Overheads

- Some variables are not defined by programmer but created by compiler during translation process.
  - Instruction labels
  - Temporary variables to store intermediate results (needed in expression parsing)
- Compilers may also generate internal tables
  - Symbol table
  - VFT (virtual function table)
- Memory required for storing data may be more than the sum of bytes required for data defined by the developer.

IF Statement

If (condition) {

Statement1

Statement2

}

Statement3

L1 and L2 are generated by compiler to ensure correct control flow.

Condition

Check condition == True

JNZ L2

Statement1
Statement2

L2: Statement3

IF..ELSE Statement

If (condition) {	Condition
Statement1	Check condition == True
}	JNZ L1
Else {	Statement1
Statement2	JMP L2
}	L1: Statement2
Statement3	L2: Statement3

While Statement

while (condition) {

Statement1

}

Statement2

L1:Condition

Check condition == True

JNZ L2

Statement1

JMP L1:

L2:Statement2

FOR Statement

for (init; cond; incr) {
Statement1
}
Statement2
	init
	L1: cond
	Check cond == True
	JNZ L2
	Statement1
	incr
	JMP L1
	L2:Statement2

Assignment

```
If( cond1 )
    Statement1
Elseif(cond2)
    Statement 2
Else
    Statement3
Endif
Statement4
```

```
Switch(value) {
case val1: stmt1; break;
case val2: stmt2; break;
case val3: stmt3; break;
default
}
```

Switch Statement

```
Switch(expr) {
Case val1:  stmt1; [break;]
Case val2: stmt2; [break;]
Case val3: stmt3; [break;]
...

Default: stmtK;
}
Next;
```

```
V = expr
If(V == val1)  goto L1;
Else if( V == val2)  goto L2;
....
Else  goto Lk;
L1: stmt1;  [goto next; ]
L2: stmt2; [goto next; ]
L3: stmt3; [goto next; ]

Lk: stmtK;
Next:
```

Assignment

Repeat

    Statement1

Until(cond)

Statement2

How would the machine code translation change when

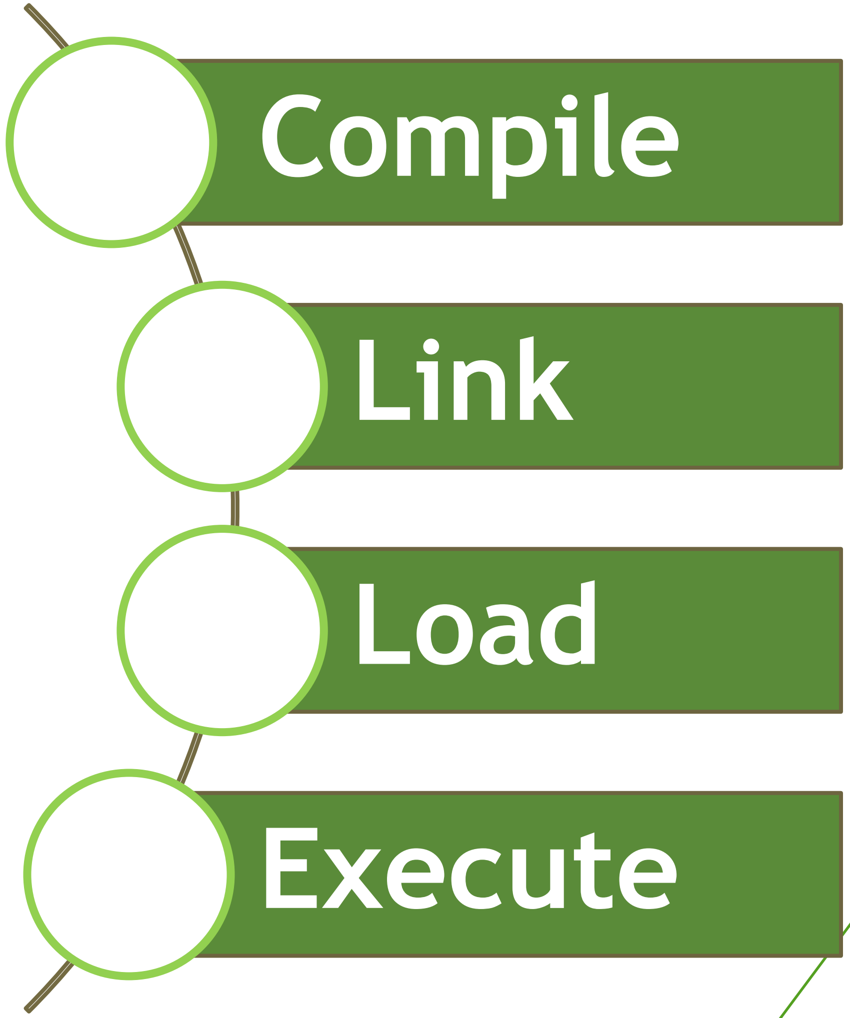
break

continue

are used?

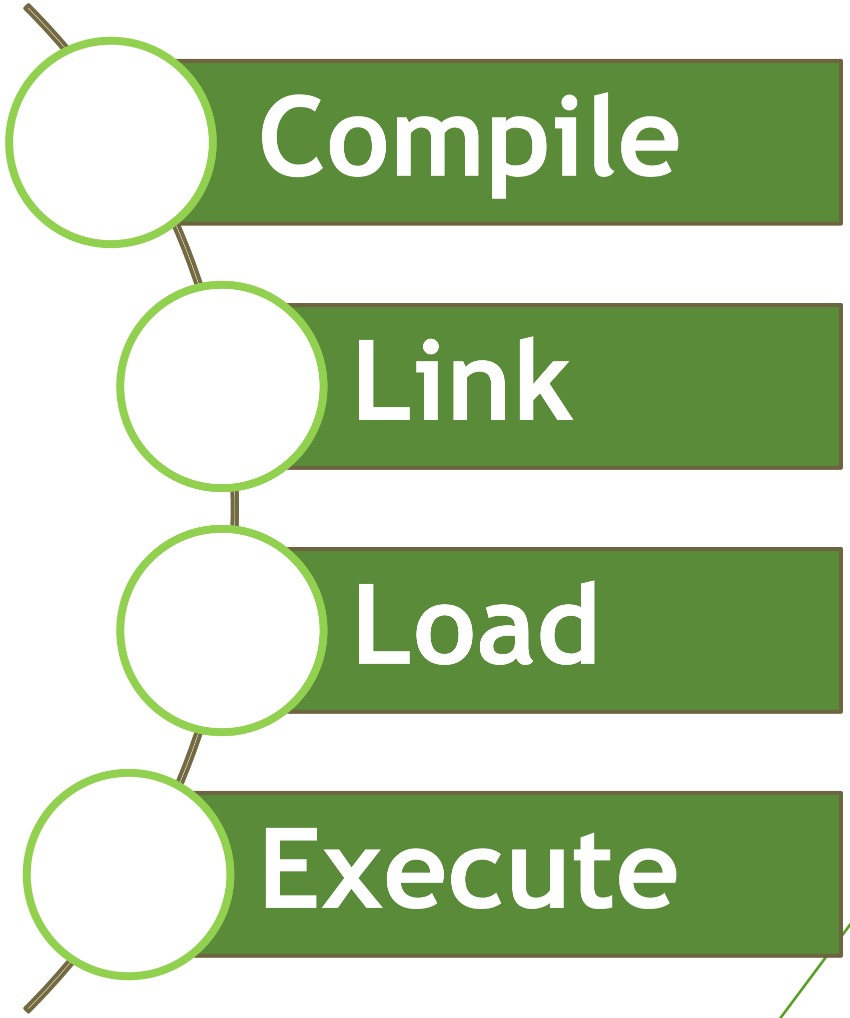
Source to Executable: Steps

- Compilation: translates a HLL program to Machine/Assembly language equivalent
- Linking: Resolves references to library functions
- Loader: Loads the program into memory
  - Absolute addresses are relocated
  - Relative addresses are retained

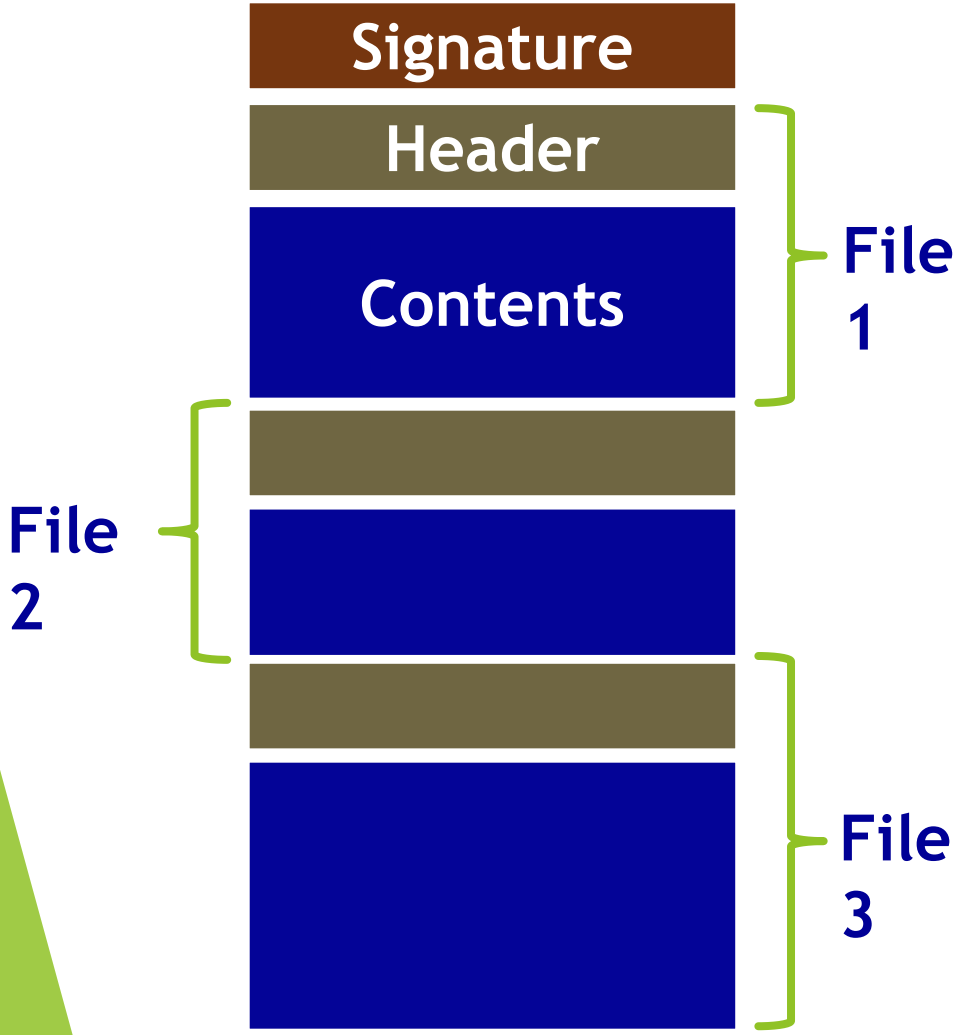


Source to Executable: Steps

- Compiled output is not machine executable
- Has unresolved references to undefined symbols
  - Library functions
  - Variables declared as **ext**
- **Linker resolves these addresses and stitches object files and library functions to create an executable image.**

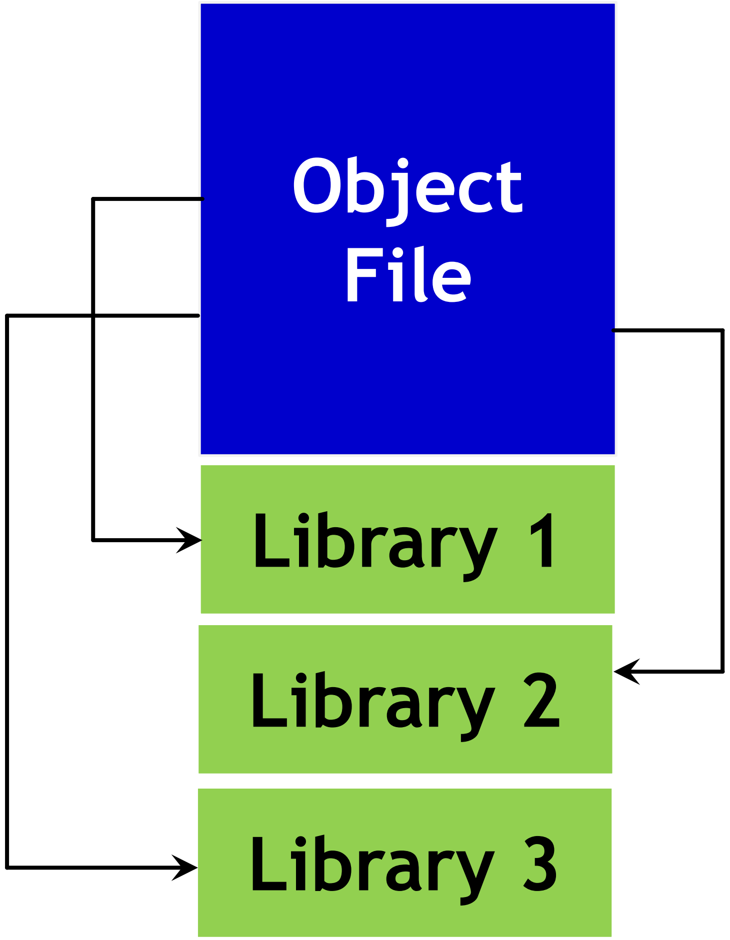


Library



- A collection of files/functions
- Preamble is
  - a signature starting with magic bytes !<arch>
  - followed by a number of files
- Each file has a header followed by its contents
  - Header is fixed length
  - Data can be arbitrarily long

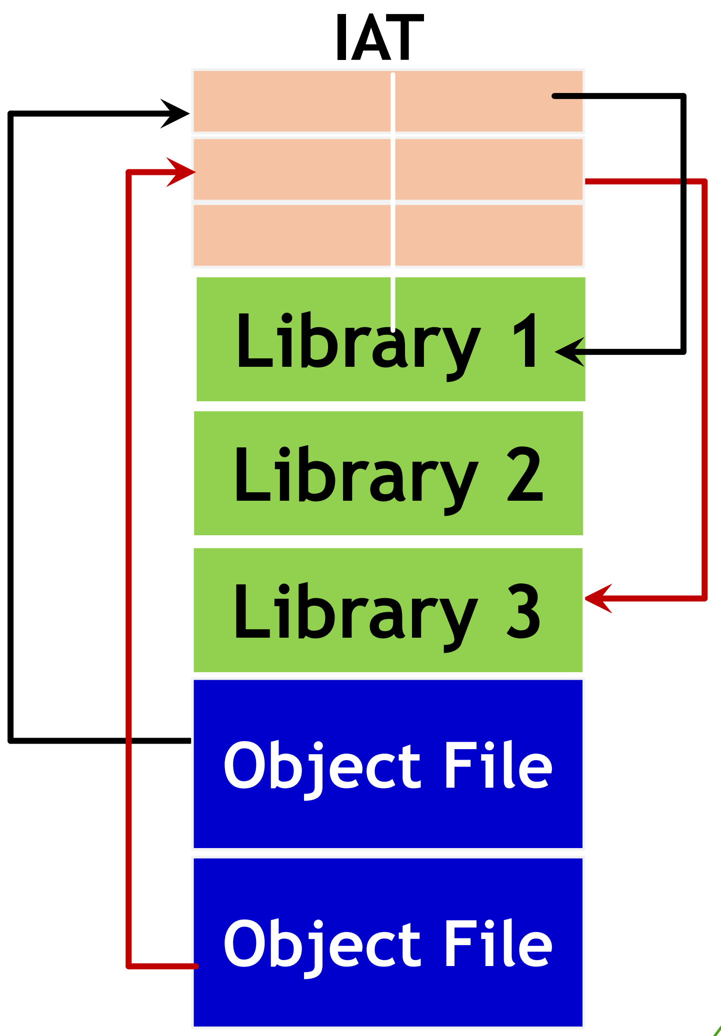
Static Linking



- Libraries added to each program
- Requires more memory space
- Multiple instances of same library in memory
- Faster as address need not be resolved at run-time

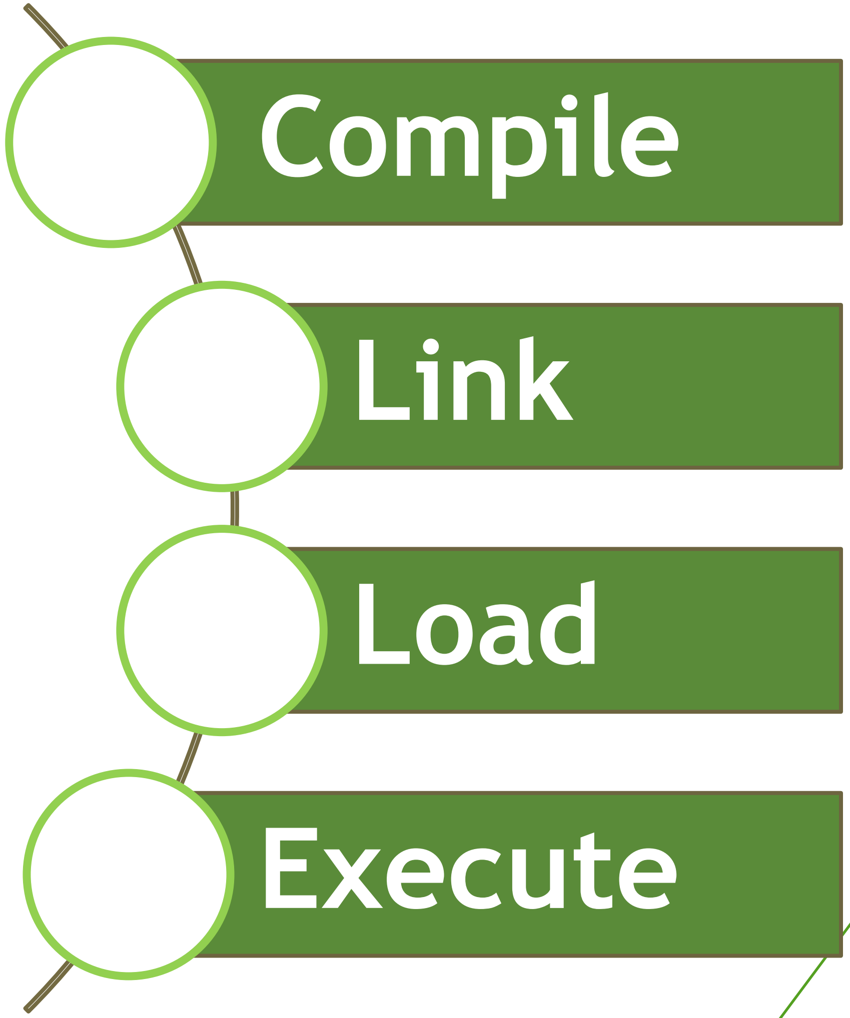
Dynamic Linking

- One instance of a library present in memory
- Library module loaded as needed
- All programs share libraries
- Call to library functions resolved at run time (dynamically linked library – DLL).
- OS keeps an import address table to locate each library module
  - Information on where a memory is loaded in memory
  - Used in resolving addresses at run-time



Source to Executable: Steps

- Linking: Resolves references to library functions
- Each library is collection of certain routines
- Multiple libraries may need to be linked
- Output contains executable file with extra information in headers
- Header information useful in loading program to memory
- Loader: Loads the program into memory
  - Absolute addresses are relocated
  - Relative addresses are retained



Address Space

- Compiler generates address space
- Virtual address space – starts with zero
  - Virtual as it is not the physical address (where program is actually loaded)
  - Processes may have similar virtual address but are mapped to different physical address
- Every instruction needs memory.
  - Number of bytes allocated for each instruction can vary
  - Even for same instruction, it could change depending on where operands are stored (Register/Memory): MOV
- Every identifier needs an address
  - Variables
  - Constants
  - Function Names: Address of first instruction (CALL)
  - Labels: Address of an Instruction (GOTO) - assembler

Why is stack needed?

- Function A consists of n instructions.
- At m^{th} instruction, A calls another function B
- Context should change from A to B
  - local variables of B should be referenced
  - IP should point to code in B
- After the function B has executed
  - A should resulme from where it left off?
- If A calls itself recursively, new context of A should be created
- Function calls: LIFO

Stack Frames

```
C( ) {
        ....
}
```

```
B( ) {
        ....
        C( );
        ....
        A( 0 );

}
```

```
A( n ) {
     if( n is non-zero)
          B( );
     endif
      ....
}
```

```
main( ) {
     A( 1 );
      .....
}
```

main( )
Stack frame

Stack Frames

```
C( ) {
        ….
}
```

```
B( ) {
        ….
        C( );
        ….
        A( 0 );

}
```

```
A( n ) {
      if( n is non-zero)
            B( );
      endif
        ….
}
```

```
main( ) {
      A( 1 );
        …..
}
```

main( ) Stack frame
A Stack frame

Stack Frames

```
C( ) {
        ....
}
```

```
B( ) {
        ....
        C( );
        ....
        A( 0 );

}
```

```
A( n ) {
     if( n is non-zero)
          B( );
     endif
        ....
}
```

```
main( ) {
     A( 1 );
       .....
}
```

main( ) Stack frame
A Stack frame
B Stack frame

Stack Frames

```
C( ) {
        ....
}
```

```
B( ) {
        ....
        C( );
        ....
        A( 0 );

}
```

```
A( n ) {
     if( n is non-zero)
          B( );
      endif
        ....
}
```

```
main( ) {
     A( 1 );
       .....
}
```

main( ) Stack frame
A Stack frame
B Stack frame
C Stack frame

Stack Frames

```
C( ) {
        ....
}
```

```
B( ) {
        ....
        C( );
        ....
        A( 0 );

}
```

```
A( n ) {
     if( n is non-zero)
          B( );
     endif
      ....
}
```

```
main( ) {
     A( 1 );
      .....
}
```

main( ) Stack frame
A Stack frame
B Stack frame

Stack Frames

```
C( ) {
        ....
}
```

```
B( ) {
        ....
        C( );
        ....
        A( 0 );

}
```

```
A( n ) {
    if( n is non-zero)
        B( );
    endif
        ....
}
```

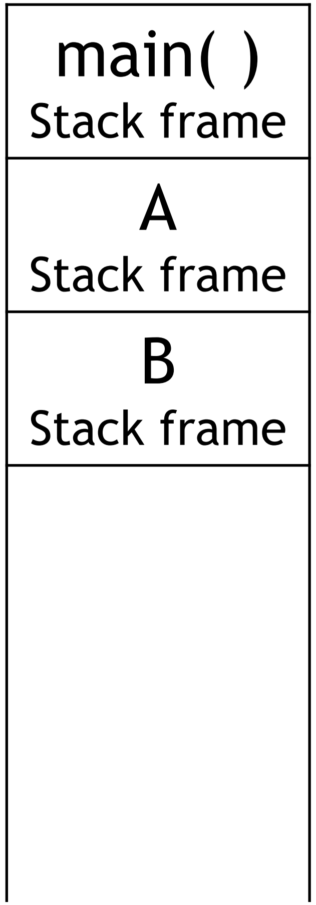
```
main( ) {
    A( 1 );
     .....
}
```

main( ) Stack frame
A Stack frame
B Stack frame
A Stack frame

Stack Frames

C( ) {
     ….
 }
	B( ) {
	     ….
	     C( );
	     ….
	     A(0);
	 }

A(n) {
     if(n is non-zero)
         B( );
     endif
     ….
 }
	main( ) {
	     A(1);
	     …..
	 }



Stack Frames

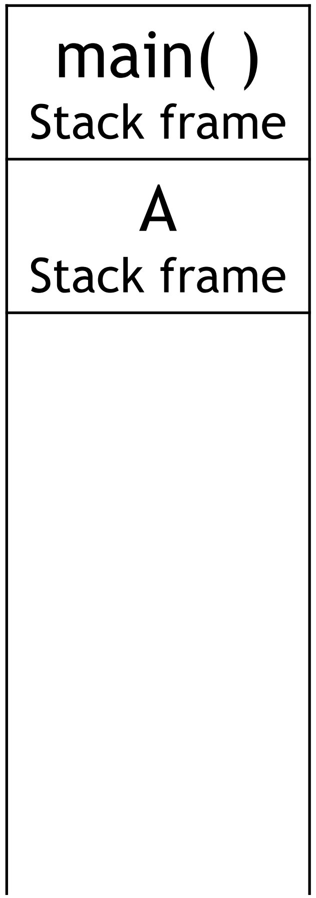
```
C( ) {
        ....
}
```

```
B( ) {
        ....
        C( );
        ....
        A( 0 );

}
```

```
A( n ) {
      if( n is non-zero)
            B( );
      endif
        ....
}
```

```
main( ) {
      A( 1 );
        .....
}
```



Stack Frames

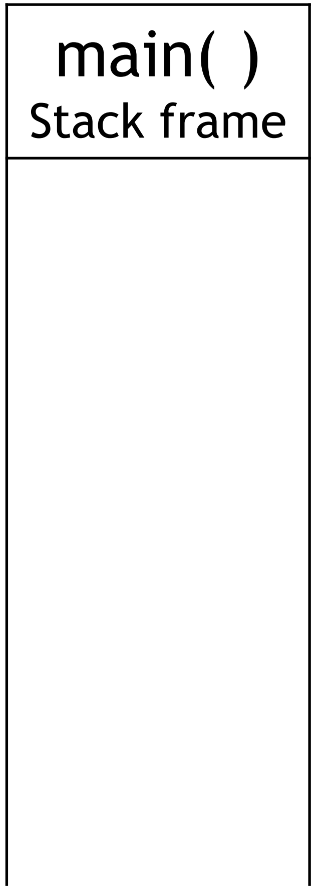
```
C( ) {
        ….
}
```

```
B( ) {
        ….
        C( );
        ….
        A( 0 );

}
```

```
A( n ) {
     if( n is non-zero)
          B( );
      endif
        ….
}
```

```
main( ) {
     A( 1 );
      …..
}
```



Stack Layout: Activation Frame

- What do we do when we call a function?
  - What data need to be stored?
  - Where do they go?

- How do we return from a function?
  - What data need to be restored?
  - Where do they come from?

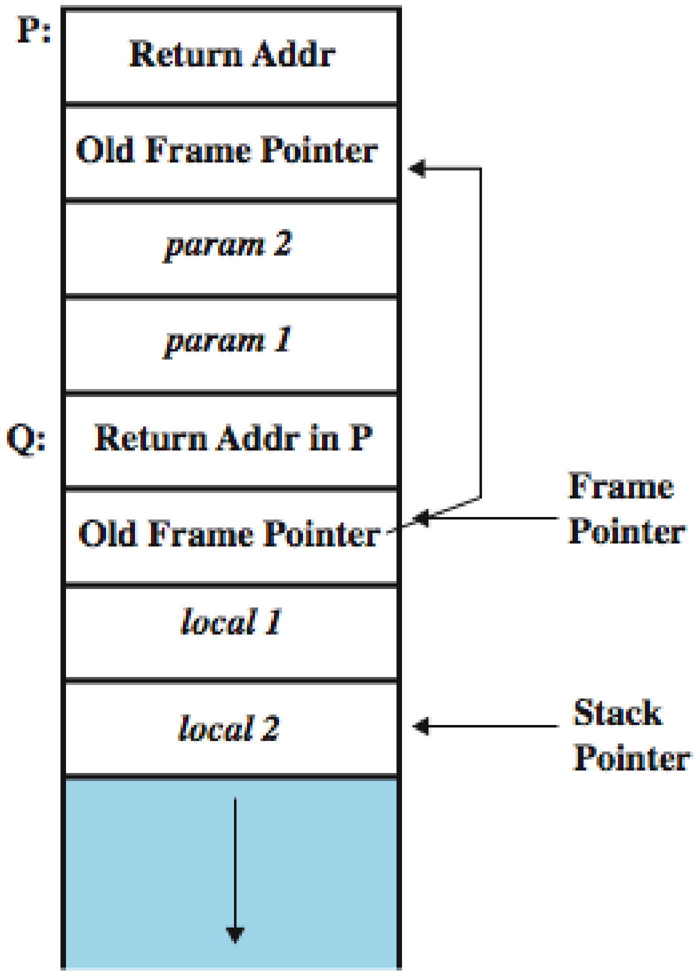
Activation Frame

- On each function call, an activation record or activation record is pushed onto stack
- Activation stack/frame consists of
  - parameters to be passed to the called function,
  - return address in callee function (instruction address where control shall return to after the called function has been executed),
  - old stack pointer value local variables of the called function,

Activation Frames (contd).

Calling function: needs a data structure to store the “return” address and parameters to be passed

Called function: needs a place to store its local variables (different for every call)



Creating New Activation Frame

- Calling function:
  - Push arguments onto the stack (in reverse order)
  - Push the address of the instruction to run after control returns to you
  - Jump to the function
- Called function:
  - Push the old frame pointer onto the stack (%ebp)
  - Set current frame pointer (%ebp) to where the end of the stack is right now (%esp)
  - Push local variables onto the stack

Activation Frame

Previous Stack Frame (main)Di
Previous Stack Frame (F1)
Function Arguments to F2 (in reverse) rightmost argument first)
Return Address (instruction address in F1)
<Previous Frame Pointer: Contents of ebp register>
Local Variables
Local Buffer Variables

- Arguments in reverse order
- Return address
- Store old frame pointer (%ebp) – so that context can be returned to callee function after this call
- Set current frame pointer (%ebp) to where the end of the stack is right now (%esp)
- Push local variables onto the stack

Different green shades show how calling and called functions contribute to activation record or stack frame. For more details, visit
https://www.cs.princeton.edu/courses/archive/spr03/cs320/notes/7-1.pdf
https://softwareengineering.stackexchange.com/questions/195385/understanding-stack-frame-of-function-call-in-c-c

Removal of Activation Frame

- Called function (to return):
  - Deallocate local variables: %esp = %ebp
  - Restore base pointer: pop %ebp
  - Jump back to where they wanted us to: %eip = (%esp)
- Calling function (on return):
  - Remove arguments from stack

Loader

- Absolute: Copies program to memory starting from a specific address (say 5000H)
- What if the program needs to be mapped to another memory location? (say 7000H)
  - All identifier addresses need to be incremented by 2000H
  - But what about relative addresses? JMP offset
- Relocatable Loader: modifies the object program by translating addresses such that the program can be loaded at an address different from the location originally specified

Process

- Process – a program in execution; process execution must progress in sequential fashion
- Process address space consists of
  - Text section: the program code
  - Current activity including program counter, processor registers
  - Stack containing temporary data, Function parameters, return addresses, local variables
  - Data section containing global variables
  - Heap for memory dynamically allocated

Process Virtual Address Space

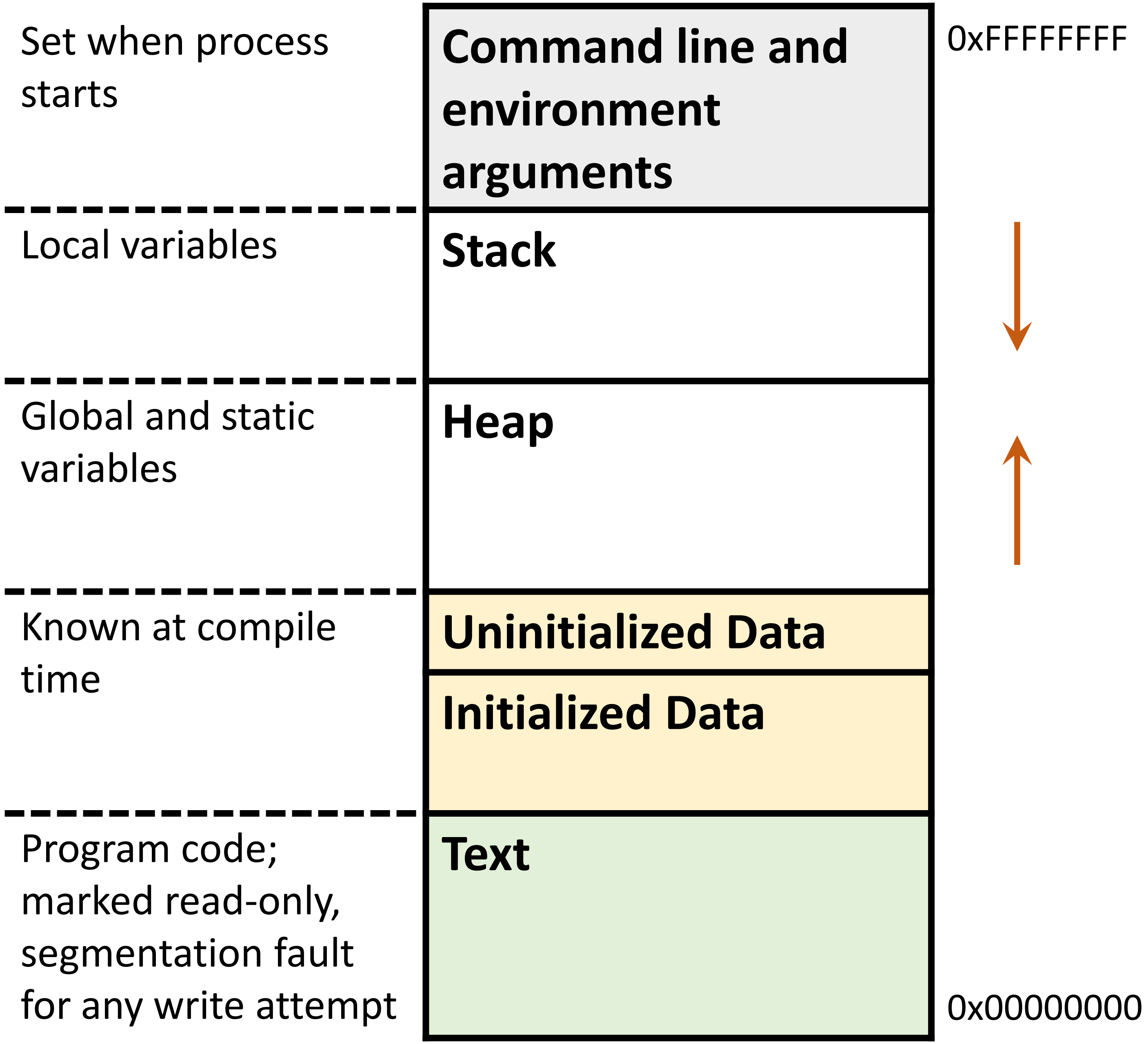
Set when process starts	**Command line and environment arguments**	0xFFFFFFFF
Local variables	**Stack**	
Global and static variables	**Heap**	
Known at compile time	**Uninitialized Data**	
	Initialized Data	
Program code; marked read-only, so any attempts to write to it will result in segmentation fault	**Text**	0x00000000

Address space

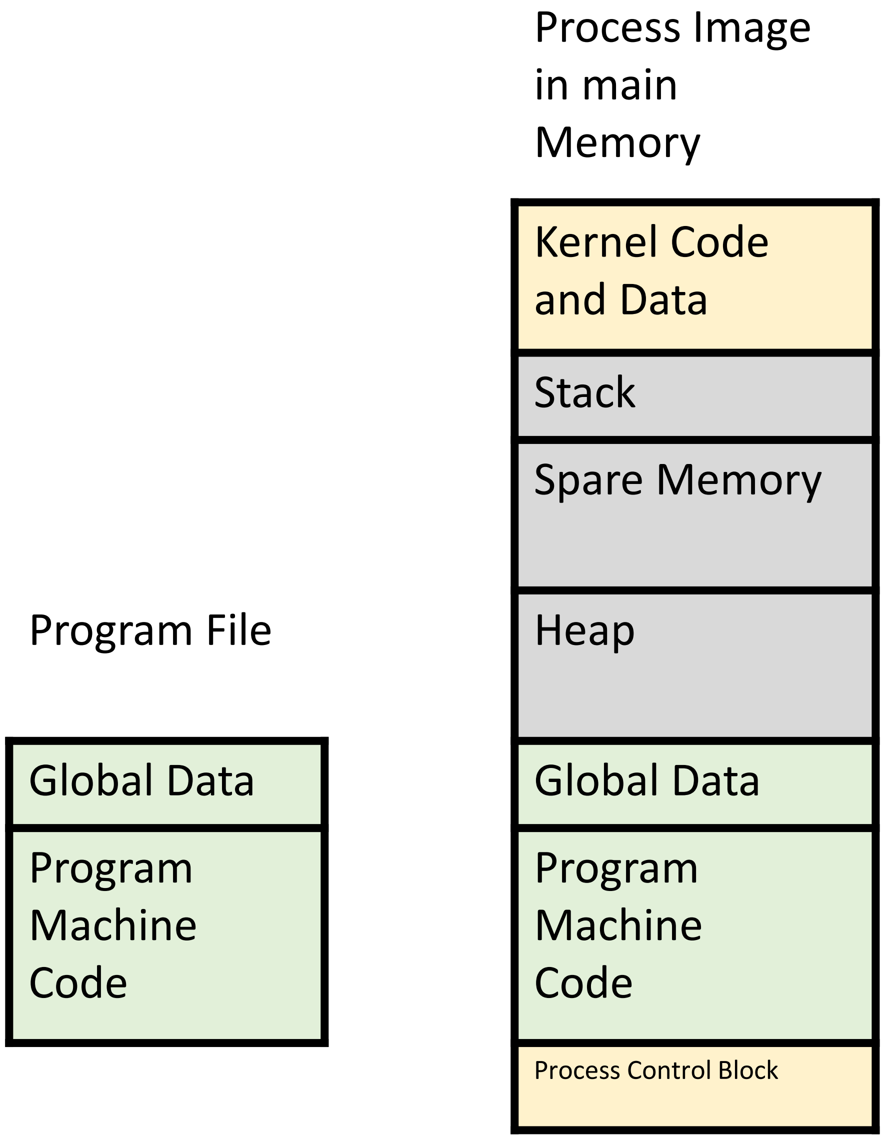
- for code instructions
  - Code Segment (Text)
- for Global and Static data – defined at compile time
  - Data Segment (BSS)
- for dynamic data – allocated at run-time through malloc( ), new( )
  - Heap
- Virtual address space always starts from Zero
  - Stack Segment

https://en.wikipedia.org/wiki/X86_memory_segmentation

Stack and heap grow in opposite directions



Programs and Processes



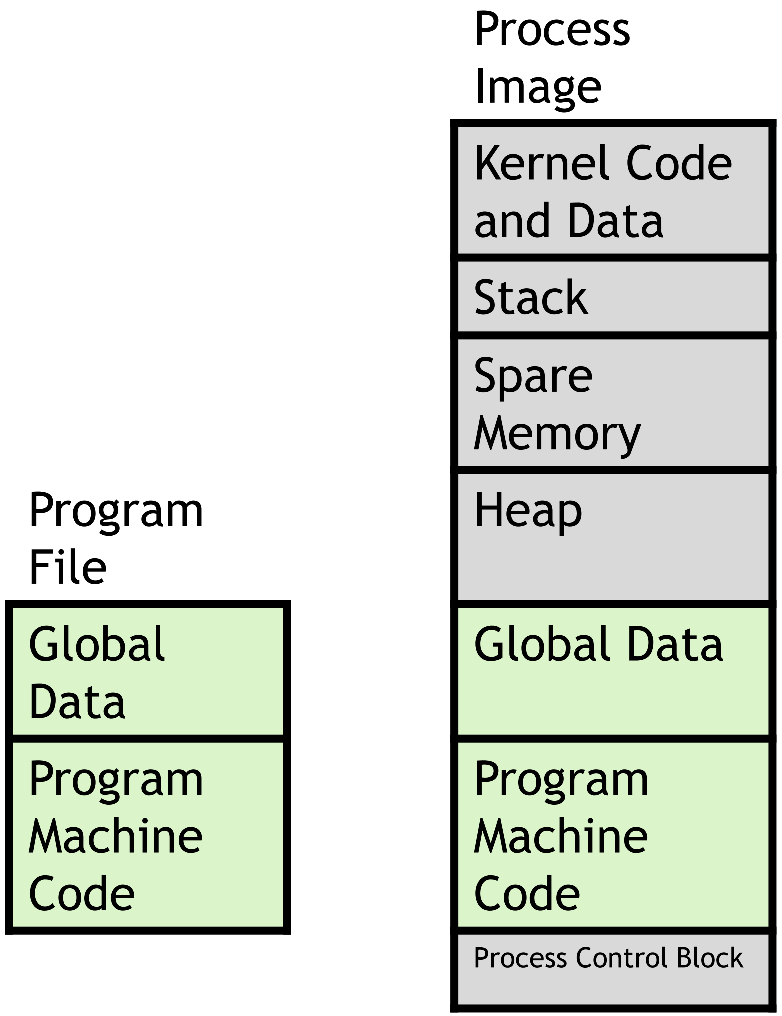
Process Address Space

- Starting address of executable program is 0000H.
- Virtual address space
- Physical address space: the memory location where program is actually loaded
- Memory Management Unit: Translates from virtual address to physical address
- Memory management
  - Paging
  - Segmentation

Context of a Process

- In multiprogramming environment, CPU time is shared between different processes.
- At time $t = t_0$, process A running.
- After a quantum (τ, $t = t_0 + \tau$), another process B is scheduled to run on CPU.
- Status of A needs to be saved/preserved so that it can be resumed from same point
  - Contents of PC and other CPU registers
  - File descriptors opened by A; resources allocated
  - Accounting information
- OS needs to maintain data structures to keep this information (context).

Program v/s Process



- In multi-processing environment, CPU time is shared between different processes.
- When CPU time allotted to a process, say A, is exhausted or process needs I/O, CPU is allocated to another process, say B.
- OS needs to ensure that when A is given back CPU, it resumes from where it left.
- For this, OS needs information (process)
  - Context: Contents of all registers including PC
  - Resources allocated to process
- This extra housekeeping information (dynamic) is what makes a process different from program.

Process Control Block

- PCB: stores all the information about a process.
- PCB is also called process descriptor.
- PCB is updated during as a process changes its state.
- One PCB per process.

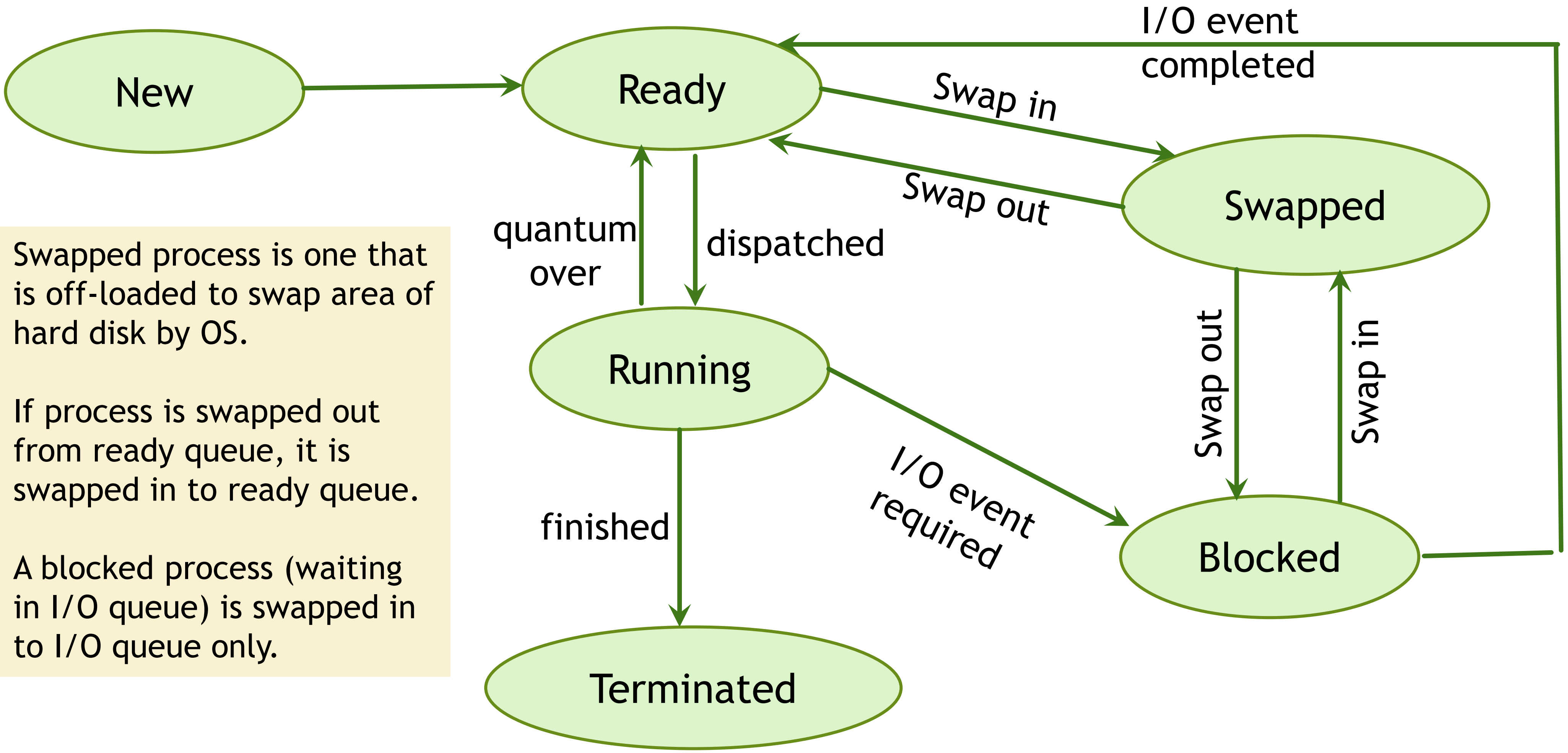
Process Control Block

- Process ID, owner, Privileges
- Process state – “ready”, “running”, "suspended",
- Child process IDs, interacting processes
- IPC information –flags, signals and messages
- Program Counter (PC) and other CPU registers
- Scheduling Information – priority, pointers to scheduling queue
- Memory Management Information–page table, memory limits, segment table
- Accounting Information–amount of CPU used for process execution, time limits, execution ID etc.
- I/O Status Information: I/O devices allocated, file descriptors

Process States

- New: request to run a program received; loader loads program image to memory; OS creates a new PCB; enters process ID in process table
- Ready: process waiting in queue to be scheduled
- Running: process allocated CPU
- Blocked/Waiting: process waiting for I/O event (put in respective queue)
- Swapped: process copied to hard disk (swap area)
- Terminated: Process has finished execution

Process State Diagram



Swapped process is one that is off-loaded to swap area of hard disk by OS.

If process is swapped out from ready queue, it is swapped in to ready queue.

A blocked process (waiting in I/O queue) is swapped in to I/O queue only.

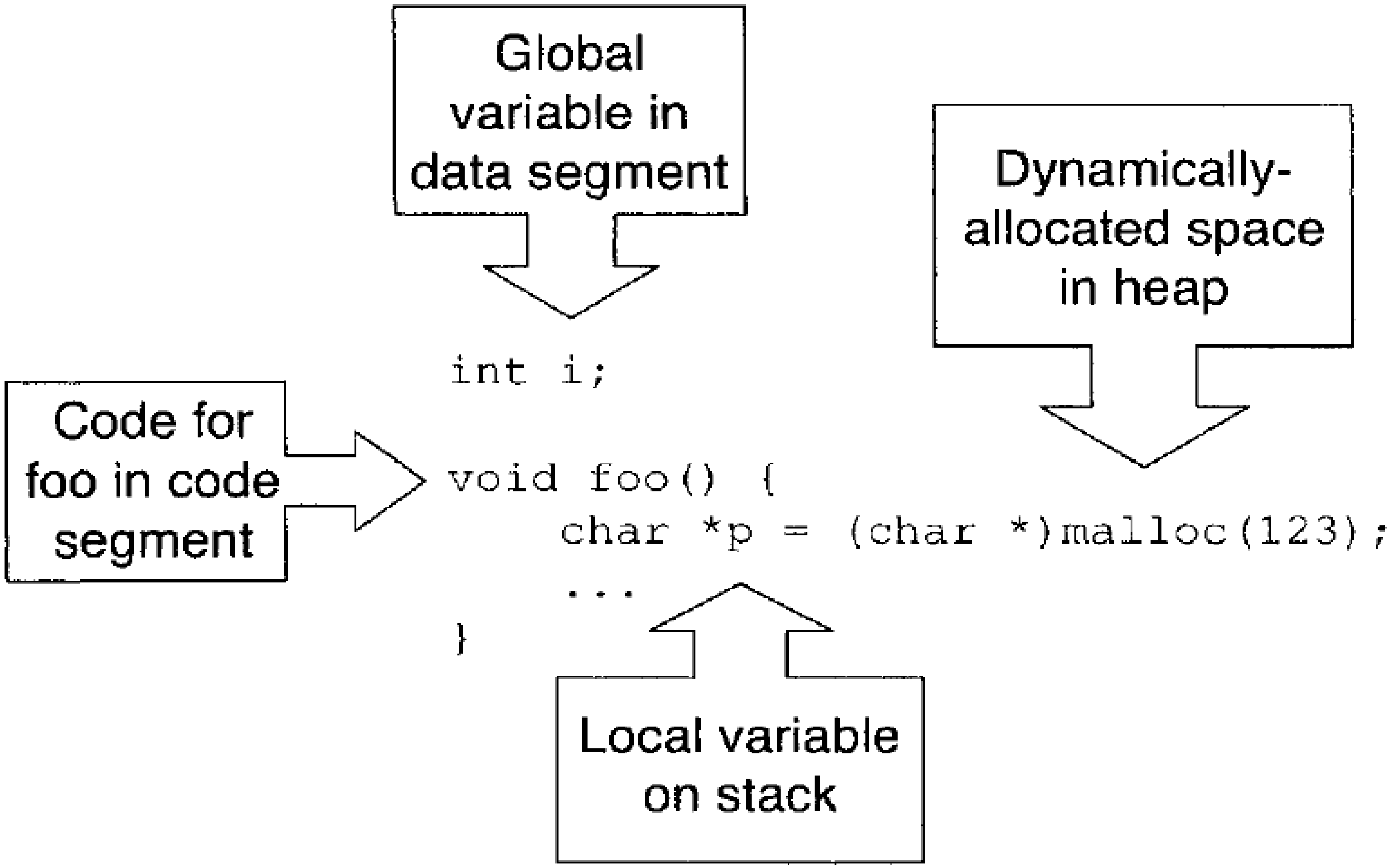
Processes and Threads

- To increase throughput of a process, threads are used.
  - for a certain functionality of the process.
  - When one thread is blocked, others can continue; the process need not be blocked.
  - Process can complete its quanta; context switches among process can reduce.
- Threads are independent sequences of instructions that are executed by the CPU without waiting for other threads.
- Threads within a process all share the same memory space, but each has its own context (processor registers) and stack.
- Any switches between threads results in saving all values in the CPU are saved in a structure called the thread context. The OS then loads the thread context of a new thread into the CPU and executes the new thread.

Segments in x86 Architecture

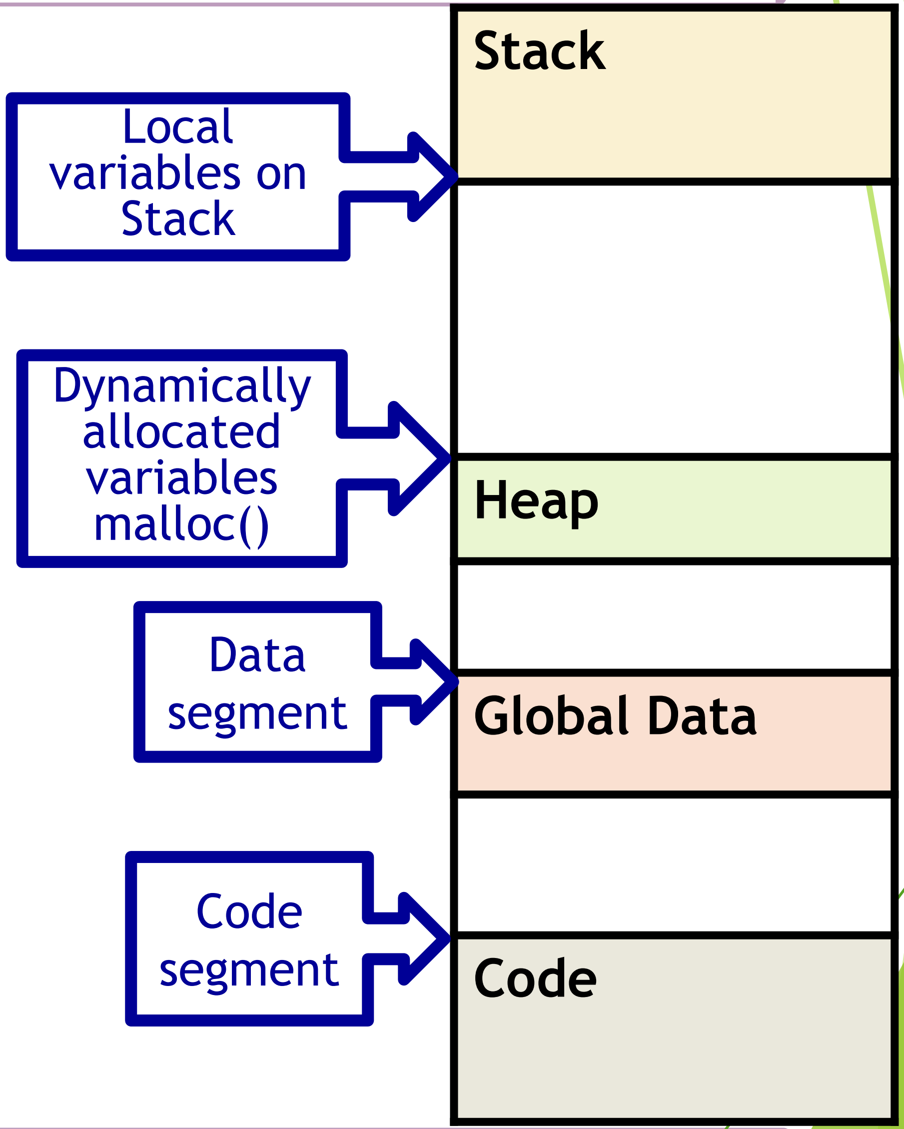
- Multi-programming requires isolation between processes
- Limit registers used for enacting isolation

Segments in x86 Architecture



Segments in x86 Architecture

- Multi-programming requires isolation between processes
  - Limit registers
  - Upper and lower bound for address
- One way is to use segments
  - Effective Address = [Segment Register] + Address
  - Contiguous memory not needed
- Code segment: instructions of the program code
- Stack segment: stack
- Data segment: global data
- Heap: Dynamically allocated data



https://en.wikipedia.org/wiki/X86_memory_segmentation

Processes v/s Threads

- A process contains one or more threads that are executed by the CPU.
- Processes also contribute to stability by preventing errors or crashes in one program from affecting other programs.
- Processes are the container for execution, but threads are what the OS executes.
- Threads are independent sequences of instructions executed by the CPU without waiting for other threads.
- A process contains one or more threads, which execute part of the code within a process.
- Threads within a process all share the same memory space, but each has its own processor registers and stack.
- Thread Context: When one thread is running, it has complete control of the CPU, or the CPU core, and other threads cannot affect the state of the CPU or core.
- When a thread changes the value of a register in a CPU, it does not affect any other threads. Before an OS switches between threads, all values in the CPU are saved in a structure called the thread context. The OS then loads the thread context of a new thread into the CPU and executes the new thread.

Thank you.