Systems 1 – CS 6013

Computer Architecture and Operating Systems

Lecture 8: Processes 2

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

SPRING 2024

*Adapted from Ryan Stutsman's slides

Lecture 8 – Topics

- Exec()
  - Creating and Killing Processes
  - Signals

Function Calls in Assembly

- What happens when you call a function in assembly? How do you call a function in assembly?
  - `call doit`
  - create a new stack frame with
    - the current IP + 1*
    - current (soon to be old) BP
    - parameters
    - variables
- What are BP, SP, IP at startup?
  - Note leaving off R for ease of reading
- Where do they live?

```
0x070 long doit( int x, int y ) {
0x071     int a = x + 2;
0x072     int b = y + 4;
0x073     long w = a + b;
0x074     return w;
      }
```

```
      const char * h = "Hello";
0x150 main() {
0x151     int z = doit(4, 8);
0x152     print( z );
      }
```

IP

CPU

ax	ip
bx	sp
cx	bp

0x9999	main	← BP
0x9998	...	
0x9997		← SP
0x9996		
0x9995		
0x9994		
0x9993		
0x9992		
0x9991		
0x9990		

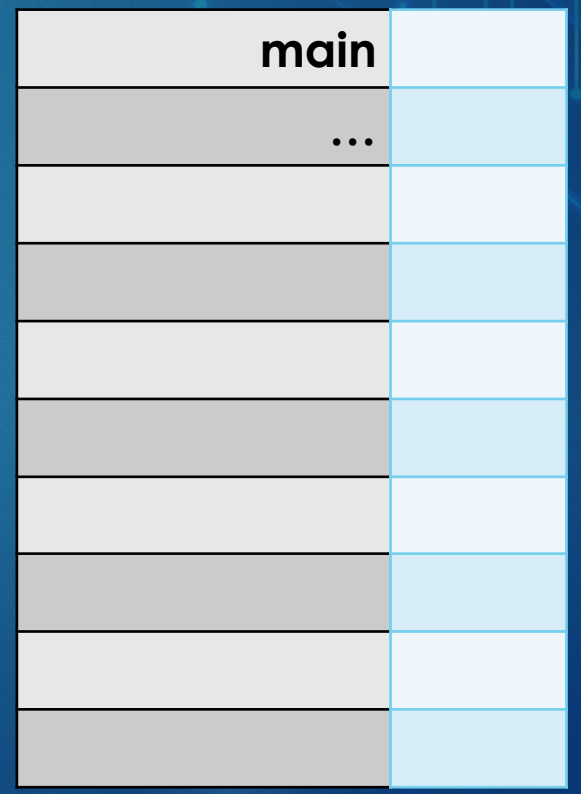
Function Calls in Assembly

- `call doit` does what?
  - saves IP (+1 instruction) to stack (This begins stack frame creation)
    - `push IP + 8` ; DONE FOR US
  - And updates IP (Jumps):
    - `mov IP, doit.` (**sort of**)
- What does our picture look like now?

```
long doit( int x, int y ) {
    int a = x + 2;
    int b = y + 4;
    long w = a + b;
    return w;
}
```

```
0x150   main() {
0x151       int z = doit(4, 8);
0x152       print( z );
        }
```

IP



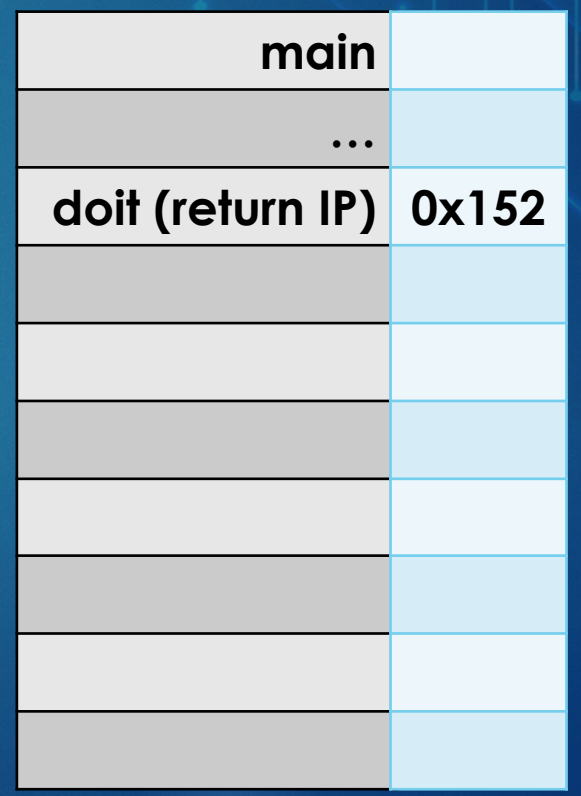
Function Calls in Assembly

- doit's prologue does what?
  - save BP…. how?
  - `push BP` ; puts in on stack

```
long doit( int x, int y ) {
    int a = x + 2;
    int b = y + 4;
    long w = a + b;
    return w;
}
```

```
0x150   main() {
0x151       int z = doit(4, 8);
0x152       print( z );
        }
```

IP



Function Calls in Assembly

- doit's prologue does what?
  - save BP…. how?
  - `push BP` ; puts in on stack
  - update BP (register in CPU)
  - `mov BP, SP ;` (Updates reg, not stack!)

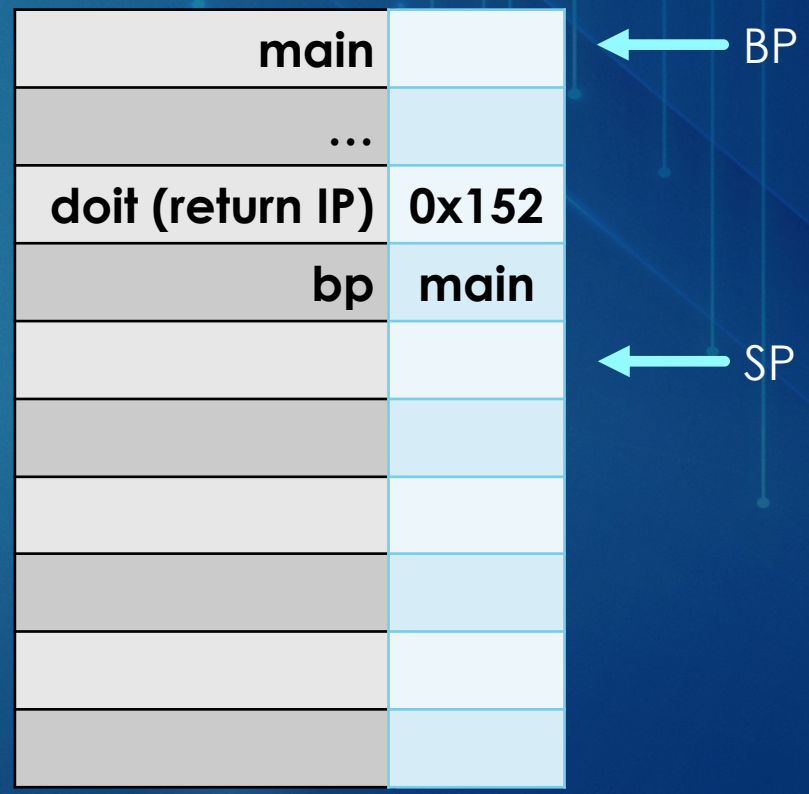
```
long doit( int x, int y ) {
    int a = x + 2;
    int b = y + 4;
    long w = a + b;
    return w;
}
```

```
0x150   main() {
0x151       int z = doit(4, 8);
0x152       print( z );
        }
```

IP

CPU

ax	ip
bx	sp
cx	bp



Function Calls in Assembly

- doit's prologue does what?
  - save BP…. how?
  - `push BP` ; puts in on stack
  - update BP (register in CPU)
  - `mov BP, SP` ; (register in CPU)
  - Next?
  - Make room for params, vars… how?
  - `sub sp, 24`

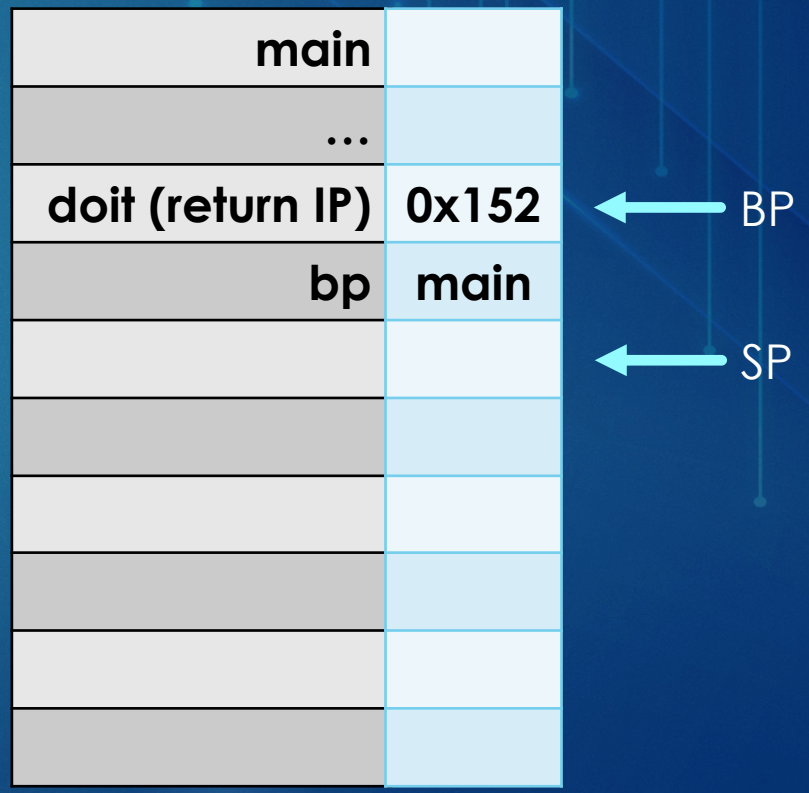
```
long doit( int x, int y ) {
    int a = x + 2;
    int b = y + 4;
    long w = a + b;
    return w;
}
```

```
0x150   main() {
0x151       int z = doit(4, 8);
0x152       print( z );
        }
```

IP

CPU

ax	ip
bx	sp
cx	bp



Function Calls in Assembly

- Clearer to do it this way?
  - `sub sp, 4 ; x`
  - `sub sp, 4 ; y`
  - `sub sp, 4 ; a`
  - `sub sp, 4 ; b`
  - `sub sp, 8 ; w`
- Now what to do with the space we just "created"?
  - Put the variables's value there… how?
  - mov [sp***], param…

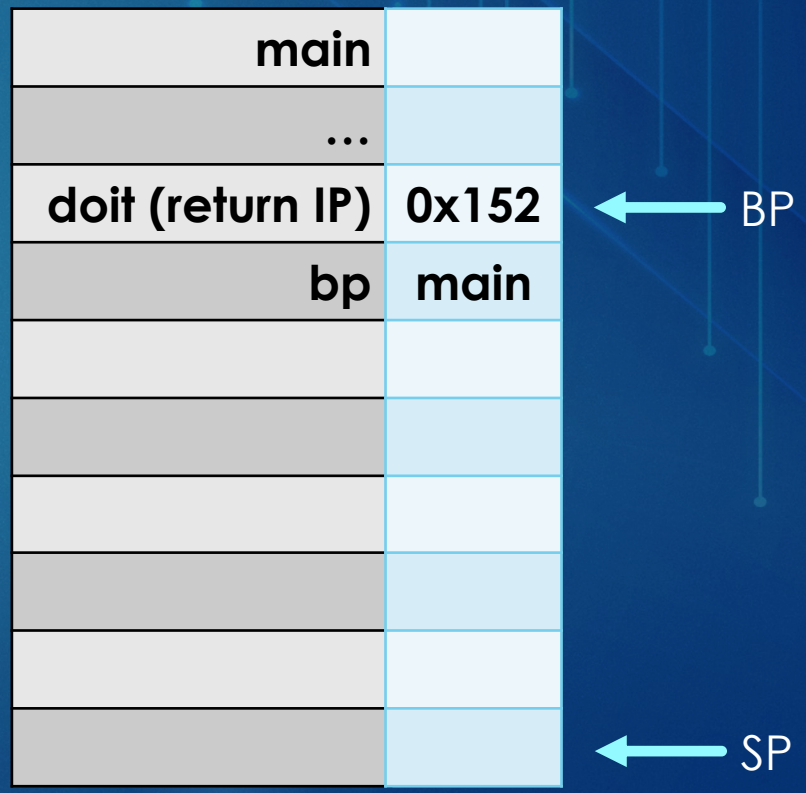
```
long doit( int x, int y ) {
    int a = x + 2;
    int b = y + 4;
    long w = a + b;
    return w;
}
```

```
0x150   main() {
0x151       int z = doit(4, 8);
0x152       print( z );
        }
```

IP

CPU

ax	ip
bx	sp
cx	bp



Function Calls in Assembly

- Remember, we are still in the prologue…
- `mov [sp+24], di`
- `mov [sp+20], si`
- `mov [sp+16], dx`
- …
- Question: Are we ***allocating*** memory?
  - Not really (not in the C++ sense)
  - We are manually using/managing memory that is already available.

```
long doit( int x, int y ) {
    int a = x + 2;
    int b = y + 4;
    long w = a + b;
    return w;
}
```

```
0x150   main() {
0x151       int z = doit(4, 8);
0x152       print( z );
        }
```

IP

CPU

ax	ip
bx	sp
cx	bp

main	
…	
doit (return IP)	0x152
bp	main
x	4
y	8
a	?
b	?
w	?

BP

SP

Function Calls in Assembly

- What happens when you return from a function in assembly? How do you that?
  - `ret`
  - But first…
  - `mov ax, [sp+8]` ; what does this do?
    - Saves "w" into return reg.
  - Now epilogue: reset SP, BP, IP
  - How?
  - `add sp, 24` ; Reset SP*

```
0x150   main() {
0x151       int z = doit(4, 8);
0x152       print( z );
        }
```

```
long doit( int x, int y ) {
    int a = x + 2;
    int b = y + 4;
    long w = a + b;
    return w;
}
```

IP

CPU

ax	ip
bx	sp
cx	bp

main	
…	
doit (return IP)	0x152
bp	main
x	4
y	8
a	?
b	?
w	?

BP

SP

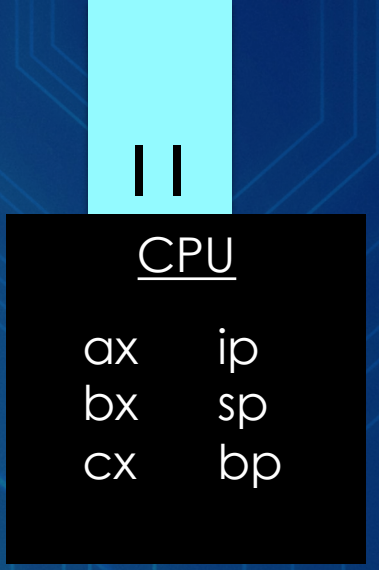
Function Calls in Assembly

- What happens when you return from a function in assembly? How do you that?
  - `pop BP` ; put "main" address into BP
    - Has side effect of also moving SP

```
long doit( int x, int y ) {
    int a = x + 2;
    int b = y + 4;
    long w = a + b;
    return w;
}
```

```
0x150   main() {
0x151       int z = doit(4, 8);
0x152       print( z );
        }
```

IP



main	
...	
doit (return IP)	0x152
bp	main
x	4
y	8
a	?
b	?
w	?

BP

SP

Returning from a Function Calls

- And finally:
- `ret`
- Which does what?
  - places the "return" address from the stack into the IP

```
int doit( int x, int y ) {
    int a = x + 2;
    int b = y + 4;
    int w = a + b;
    return w;
}
```

```
0x150   main() {
0x151       int z = doit(4, 8);
0x152       print( z );
        }
```

IP

CPU

ax	ip
bx	sp
cx	bp

main	
...	
doit (return IP)	0x152
bp	main
x	4
y	8
a	?
b	?
w	?

BP

SP

Returning from a Function Calls

- Leaving the original function (in this case `main()`) to continue where it left off.
- Notice the BP, and SP are back to pointing at the correct locations in memory (on the stack)
- Why is 0x152, main, 4, 8, ?, ? still on the stack?
  - Garbage left there that will be overwritten the next time a function is called.

```
int doit( int x, int y ) {
    int a = x + 2;
    int b = y + 4;
    int w = a + b;
    return w;
}
```

```
0x150   main() {
0x151       int z = doit(4, 8);
0x152       print( z );
        }
```

CPU

ax	ip
bx	sp
cx	bp

main	
...	
doit (return IP)	0x152
bp	main
x	4
y	8
a	?
b	?
w	?

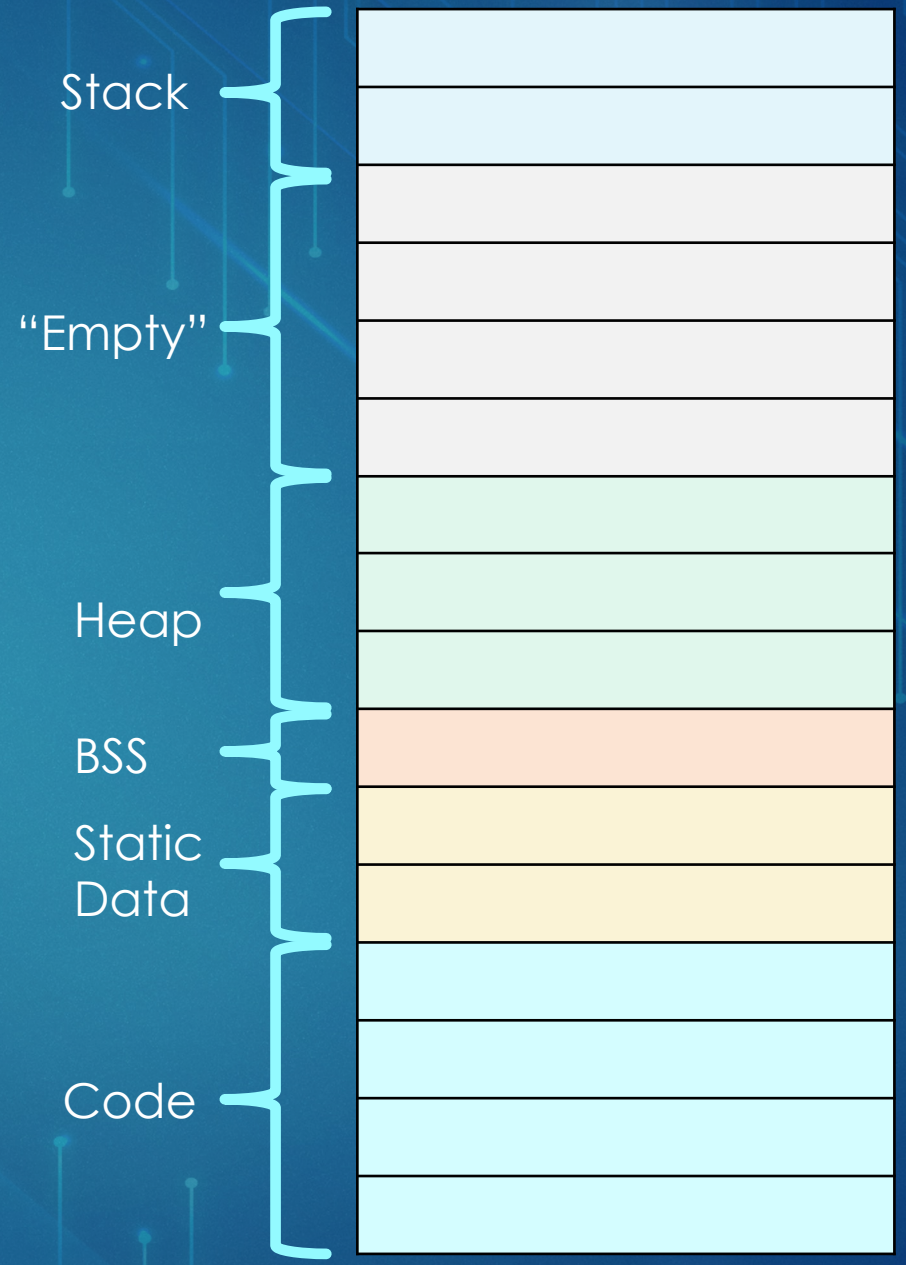
BP

SP

Address Space

- The memory for this process (its Address Space) looks like this (using information from previous slides):
  - Sections?
  - Values?
- Update your picture assuming `main()` contains this code:

```
// Note, the following code is not
// 100% syntactically correct…
char * test = malloc(6);
*test = "world";
```

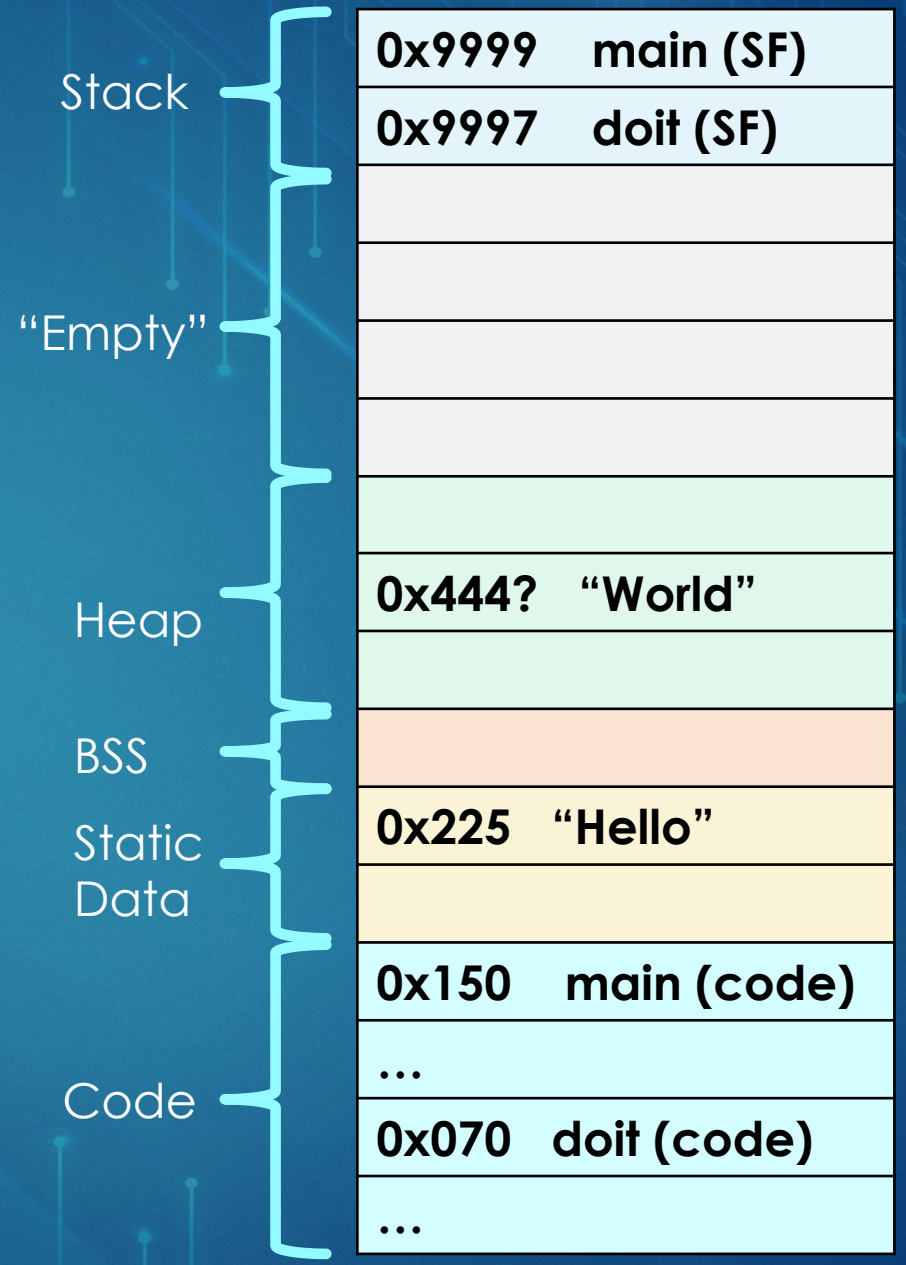


Address Space

- The memory for this process (its Address Space) looks like this (using information from previous slides):
  - Sections?
  - Values?
  - What are IP, BP, and SP (in middle of doit?)
- Update your picture assuming `main()` contains this code:

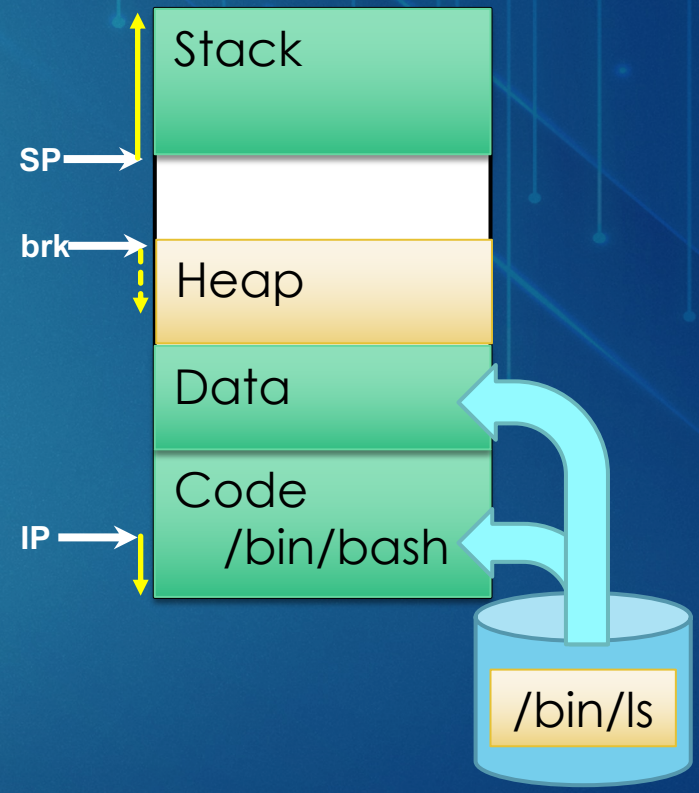
```
// Note, the following code is not
// 100% syntactically correct…
char * test = malloc(6);
*test = "world";
```

SF – Stack Frame



exec(): Run Another Program

- Replaces process with another program
- Loads program from filesystem
  - Replaces code, data, bss
  - Put argv on stack; reset sp
  - Release heap memory
  - Reset ip to main (really _start)
- exec() only returns to caller if failed
  - Otherwise process is now in a new main



Why Separate fork and exec?

- Lots of parameters on creating a process
  - Shell may want to
    - redirect output of children
    - change child environment
    - change child working directory
    - run child as a different user
- Hard to create simple, expressive-enough API
- Separation allows policy to be expressed in parent's program but in child's process

exec() demo

Termination: `exit()`, `kill()`

- When process dies, OS reclaims resources
  - Record exit status in PCB
  - Close files, sockets
  - Free memory
  - Free (nearly) all kernel structures
- Process terminates with exit()
- Process terminates another with kill()

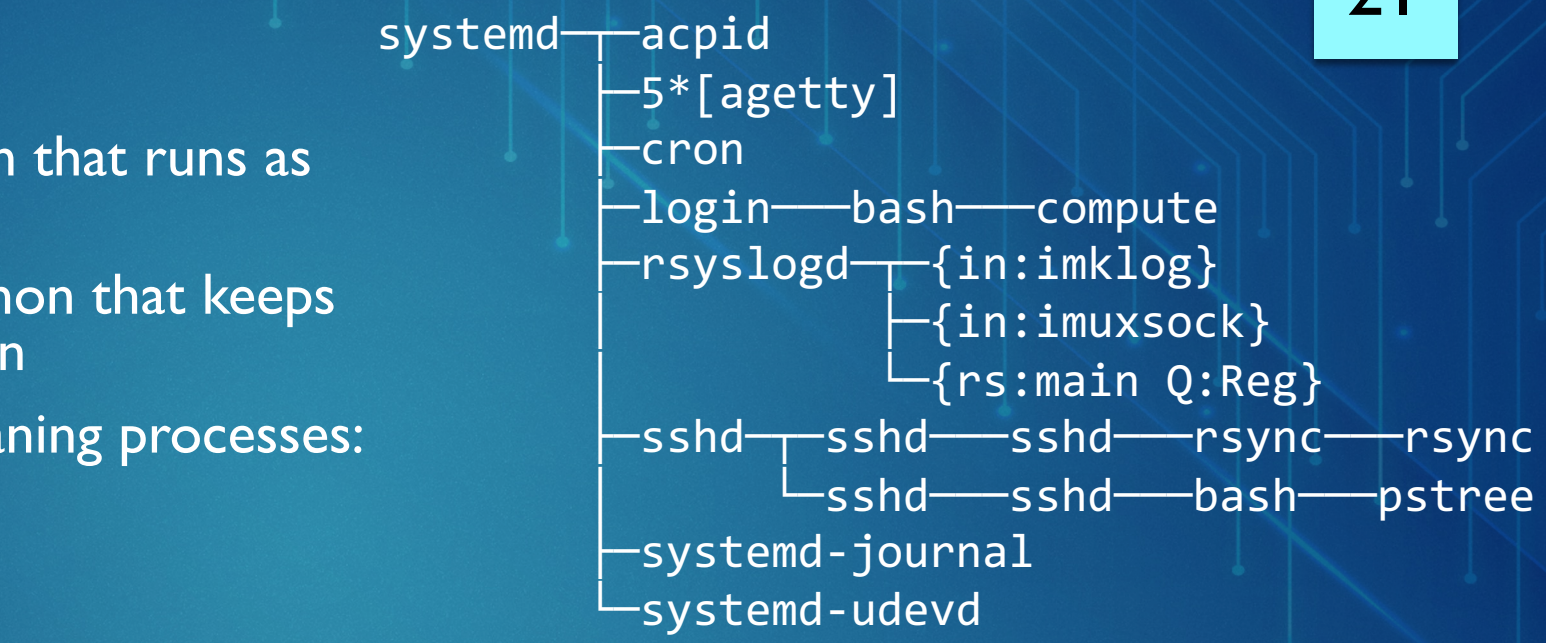
```
int main( int argc, char* argv[]) {
    pid_t pid = fork();
    if (pid == 0) {
      sleep(10);
      printf("Child exiting!\n");
      exit(0);
    }
    else {
      sleep(5);
      if( kill( pid, SIGKILL) != -1 ){
        printf("Sent kill!\n");
      }
    }
}
```

Orphans and Zombies

- Parent wait() on child returns status
- Must keep around PCB with status after child exit
- **Zombie**: exited process with uncollected status
- Parent exits before child? **Orphaned**
  - init daemon adopts orphans
  - Collects and discards status of reparented children
  - Could be handed off to a "subreaper" instead of init
  - A useful way to create and start daemons.

Daemons

- Daemon- computer program that runs as background process
- First process (**init**) is a daemon that keeps running while computer is on
- Can start daemons by orphaning processes:
  - fork() twice
  - Terminate child
  - Grandchild process is adopted by **init**
- Many ways of starting daemons
- Single init daemon has been mostly replaced by systemd (right), which contains 69 programs (including **init**)



~ Fin ~