Roadmap

C:
```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:
```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

Assembly language:
```
get_mpg:
pushq   %rbp
movq    %rsp, %rbp
...
popq    %rbp
ret
```

Machine code:
```
0111010000011000 100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:
```

Windows 8
Mac
```

Computer system:
```

CPU
Memory
Storage
```

Data & addressing
Integers & floats
Machine code & C
x86 assembly
programming
Procedures &
stacks
Arrays & structs
Memory & caches
Exceptions &
processes
Virtual memory
Memory allocation
Java vs. C
What is a process?

- Why are we learning about processes?
  - Processes are another abstraction in our computer system – the process abstraction provides an interface between the program and the underlying CPU + memory.

- What do processes have to do with exceptional control flow (previous lecture)?
  - Exceptional control flow is the mechanism that the OS uses to enable multiple processes to run on the same system.

- What is a program? A processor? A process?
Processes

Definition: A *process* is an instance of a running program
- One of the most important ideas in computer science
- Not the same as “program” or “processor”

Process provides each program with **two key abstractions**:
- Logical control flow
  - Each process seems to have exclusive use of the CPU
- Private virtual address space
  - Each process seems to have exclusive use of main memory

Why are these illusions important?

How are these illusions maintained?
- Process executions interleaved (multi-tasking)
- Address spaces managed by virtual memory system – next course topic
Concurrent Processes

- Two processes run *concurrently* (are concurrent) if their instruction executions (flows) overlap in time.
- Otherwise, they are *sequential*.
- Examples:
  - Concurrent: A & B, A & C
  - Sequential: B & C
User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time
  - CPU only executes instructions for one process at a time
- However, we can think of concurrent processes as executing in parallel

```
<table>
<thead>
<tr>
<th></th>
<th>Process A</th>
<th>Process B</th>
<th>Process C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Context Switching

- Processes are managed by a shared chunk of OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of a user process

- Control flow passes from one process to another via a *context switch*... (how?)
Creating New Processes & Programs

- **fork-exec model:**
  - `fork()` creates a copy of the current process
  - `execve()` replaces the current process’ code & address space with the code for a different program

- **fork() and execve() are system calls**
  - Note: process creation in Windows is slightly different from Linux’s fork-exec model

- **Other system calls for process management:**
  - `getpid()`
  - `exit()`
  - `wait()` / `waitpid()`
fork: Creating New Processes

- `pid_t fork(void)`
  - creates a new process (child process) that is identical to the calling process (parent process)
  - returns 0 to the child process
  - returns child’s process ID (`pid`) to the parent process

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

- `fork` is unique (and often confusing) because it is called *once* but returns *twice*
Understanding fork

Process n

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Child Process m

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Which one is first?
Fork Example

- Parent and child both run the same code
  - Distinguish parent from child by return value from `fork()`
  - Which runs first after the `fork()` is undefined

- Start with same state, but each has a `private` copy
  - Same variables, same call stack, same file descriptors...

```c
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```
Fork-Exec

- **fork-exec model:**
  - `fork()` creates a copy of the current process
  - `execve()` replaces the current process’ code & address space with the code for a different program
    - There is a whole family of *exec* calls – see `exec(3)` and `execve(2)`

```c
// Example arguments: path="/usr/bin/ls",
void fork_exec(char *path, char *argv[])
{
    pid_t pid = fork();
    if (pid != 0) {
        printf("Parent: created a child %d\n", pid);
    } else {
        printf("Child: exec-ing new program now\n");
        execv(path, argv);
    }
    printf("This line printed by parent only!\n");
}
```
Exec-ing a new program

Very high-level diagram of what happens when you run the command "ls" in a Linux shell:
**execve**: Loading and Running Programs

- **int execve**(
  - char *filename,
  - char *argv[],
  - char *envp[]
)

- **Loads and runs in current process:**
  - Executable **filename**
  - With argument list **argv**
  - And environment variable list **envp**
    - Env. vars: “name=value” strings (e.g. “PWD=/homes/iws/pjh”)

- **execve does not return** (unless error)

- **Overwrites code, data, and stack**
  - Keeps pid, open files, a few other items
exit: Ending a process

- **void exit(int status)**
  - Exits a process
    - Status code: 0 is used for a normal exit, nonzero for abnormal exit
  - **atexit()** registers functions to be executed upon exit

```c
void cleanup(void) {
    printf("cleaning up\n");
}

void fork6() {
    atexit(cleanup);
    fork();
    exit(0);
}
```
Zombies

■ Idea
  ▪ When process terminates, it still consumes system resources
    ▪ Various tables maintained by OS
  ▪ Called a “zombie”
    ▪ A living corpse, half alive and half dead

■ Reaping
  ▪ Performed by parent on terminated child
  ▪ Parent is given exit status information
  ▪ Kernel discards process

■ What if parent doesn’t reap?
  ▪ If any parent terminates without reaping a child, then child will be reaped by \texttt{init} process (pid == 1)
  ▪ But in long-running processes we need \textit{explicit} reaping
    ▪ e.g., shells and servers
`wait`: Synchronizing with Children

- `int wait(int *child_status)`
  -Suspends current process (i.e. the parent) until one of its children terminates
  -Return value is the `pid` of the child process that terminated
    -On successful return, the child process is reaped
  -If `child_status != NULL`, then the int that it points to will be set to a status indicating why the child process terminated
    -There are special macros for interpreting this status – see `wait(2)`

- If parent process has multiple children, `wait()` will return when *any* of the children terminates
  -`waitpid()` can be used to wait on a specific child process
wait Example

```c
void fork_wait() {
    int child_status;
    pid_t child_pid;

    if (fork() == 0) {
        printf("HC: hello from child\n");
    } else {
        child_pid = wait(&child_status);
        printf("CT: child %d has terminated\n", child_pid);
    }
    printf("Bye\n");
    exit(0);
}
```
Process management summary

- **fork** gets us two copies of the same process (but **fork()** returns different values to the two processes)

- **execve** has a new process substitute itself for the one that called it
  - Two-process program:
    - First **fork()**
    - if (pid == 0) { //child code } else { //parent code }
  - Two different programs:
    - First **fork()**
    - if (pid == 0) { **execve()** } else { //parent code }
    - Now running two completely different programs

- **wait**/**waitpid** used to synchronize parent/child execution and to reap child process
Summary

■ Processes
  - At any given time, system has multiple active processes
  - Only one can execute at a time, but each process appears to have total control of the processor
  - OS periodically “context switches” between active processes
    - Implemented using *exceptional control flow*

■ Process management
  - fork-exec model
Fork Example #2

- Both parent and child can continue forking

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

L0  By e
```
L1  

By e
```

L0  
```
L1  

By e
```
Fork Example #3

- Both parent and child can continue forking

```c
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```

Diagram of execution:
```
    L0
     |   L1  L2
     |     |   L2
     |     |   Bye
     |     Bye
     Bye
```
Both parent and child can continue forking

```c
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```
Both parent and child can continue forking

```c
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```
Zombie Example

```
void fork7()
{
   if (fork() == 0) {
      /* Child */
      printf("Terminating Child, PID = %d\n", getpid());
      exit(0);
   } else {
      printf("Running Parent, PID = %d\n", getpid());
      while (1)
         ; /* Infinite loop */
   }
```

- **ps** shows child process as “defunct”

- Killing parent allows child to be reaped by **init**

```
Eye 25
```
Non-terminating Child Example

```c
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n", getpid());
        exit(0);
    }
}
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely
wait() Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```c
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```
waitpid(): Waiting for a Specific Process

- **waitpid(pid, &status, options)**
  - suspends current process until specific process terminates
  - various options (that we won’t talk about)

```c
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```