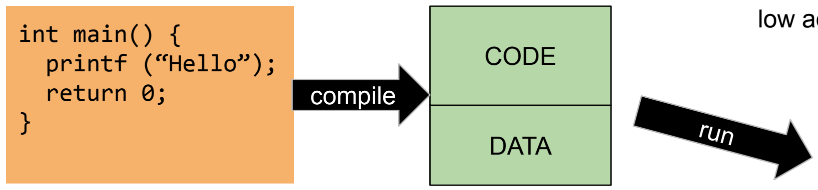
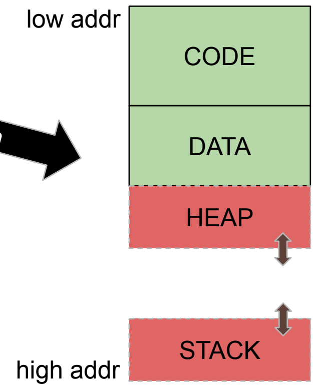
Unix Process Management

Process ≠ Program

Source ⇒ Binary Executable ⇒ Process (Image)





- *Code and data sections are generated at **compile-time***
  - *Data whose type and size can be determined at compile time*
- *Heap and stack are created at **runtime***
- ***Image snapshot***

Data Section vs. Heap vs. Stack

- Data section
  - Global variables (lifetime is NOT tied to any function, but tied to the entire program)
- Heap
  - Dynamically allocated memory (`malloc()`, Java `new`, C++ `new`)
- Stack
  - **Local** variables (lifetime is tied to the function lifetime)
  - C **fixed length** arrays
  - **Function Parameters**
  - **Return addresses**

Memory Allocation: (PollEv.com)

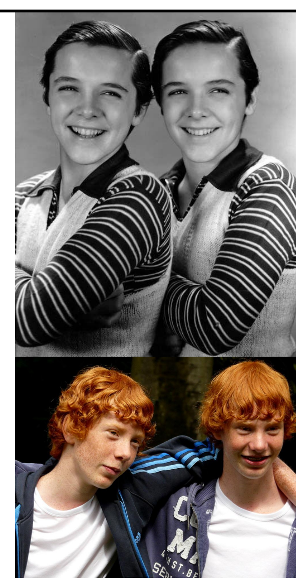
```
float here;

int main() {
  int val;
  float *numbers;
  numbers = calloc (100, sizeof(float));
  for (........) {

  }
  free (numbers);
  return 0;
}
```

Data section, stack, or heap?

- `here` is allocated in ______________
- `val` is allocated in ____________
- `numbers` is allocated in ___________
- The 400 bytes from calloc() is allocated in _________________

Assume 4-byte floats

fork()



- Create a new (child) process using the **current copy** of the parent image
  - Parent and child processes are like **twins**!
- **At the time of fork()**
  - Code section: **exact duplicate** of each other
  - Heap section: **exact duplicate** of each other (most likely)
- **After return from fork()**
  - *Data (or Stack) section may differ by a few bytes*
- Thereafter, the two images are **independent/unrelated**
  - Parent and child **share NOTHING (***not actually TRUE, further explanation later***)**

Fork()

```
/* parent */
int main() {
  printf ("Begin\n");
  fork();
  printf ("PID = %d\n, getpid());
  printf ("End\n");
  return 0;
}
```


Fork()

```
/* parent */
int main() {
  printf ("Begin\n");
  fork();
  printf ("PID = %d\n, getpid());
  printf ("End\n");
  return 0;
}
```

Fork()

a new process is created, DUPLICATING the parent process image

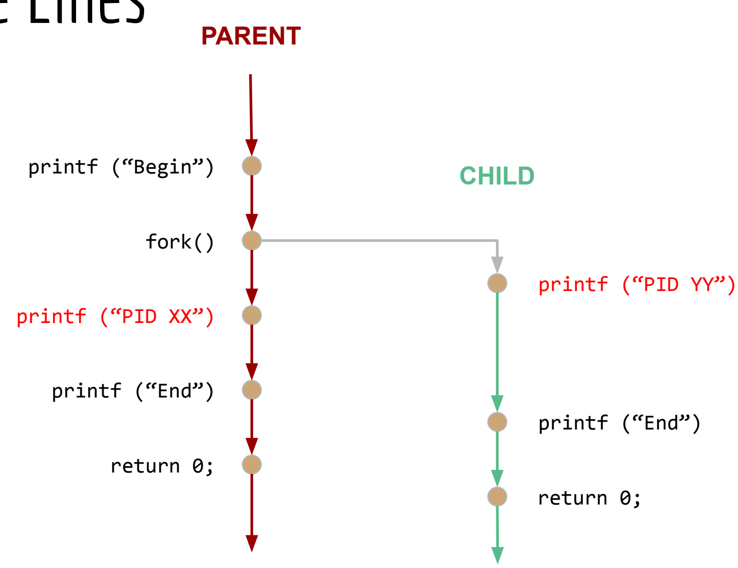
```
/* parent */
int main() {
  printf (“Begin\n”);
  fork();
  printf (“PID = %d\n, getpid());
  printf (“End\n”);
  return 0;
}
```

```
/* child */
int main() {
  printf (“Begin\n”);
  fork();
  printf (“PID = %d\n, getpid());
  printf (“End\n”);
  return 0;
}
```

At this point, both parent and child will run *independently* competing for the same CPU

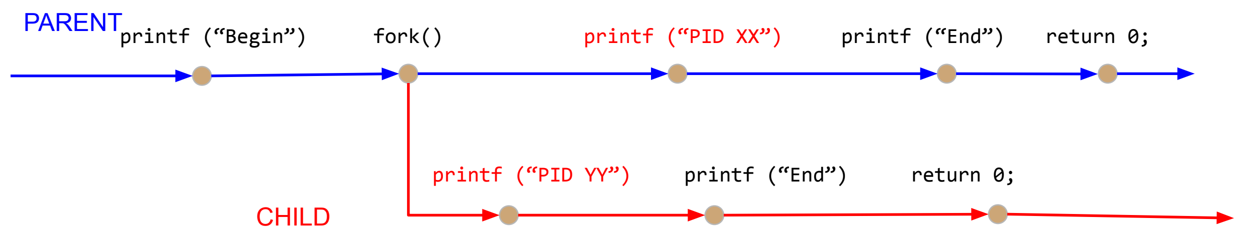
Parent & Child Time Lines

```
/* parent */
int main() {
  printf (“Begin\n”);
  fork();
  printf (“PID = %d\n, getpid());
  printf (“End\n”);
  return 0;
}
```



Parent & Child Time Lines (Horizontal)

```
/* parent */
int main() {
  printf ("Begin\n");
  fork();
  printf ("PID = %d\n, getpid());
  printf ("End\n");
  return 0;
}
```



Review

1. fork() creates a new process (child) by copying the current image (parent)
   a. The child image is **EXACT DUPLICATE** of the parent image
2. **No** code/data/heap/stack are shared between parent and child
   a. When one modifies its data/heap/stack the other won't see it
3. Both processes (parent & child) will **run independently** and compete for the same CPU(s) on your system
   a. We never know the relative order of execution ACROSS the two processes
   b. We only know the relative order of execution WITHIN each process
4. **Both processes** will resume execution to the **next statement after fork()**

True / False?

The OS inspects each instruction of a program before it runs on the CPU

Fork()

```
/* parent */
int main() {
  printf ("Begin\n");
  pid_t who = fork();
  if (who == 0)
     printf ("Mug %d\n, getpid());
  else {
     printf ("Cup %d\n, who);
     printf ("Bowl %d\n", getpid());
  }

  printf ("End\n");
  return 0;
}
```

Fork()

```
/* parent */
int main() {
  printf (“Begin\n”);
  pid_t who = fork();
  if (who == 0)
     printf (“Mug %d\n, getpid());
  else {
     printf (“Cup %d\n, who);
     printf (“Bowl %d\n”, getpid());
  }

  printf (“End\n”);
  return 0;
}
```

`fork()` return value:

- ZERO (in child process)
- child PID (in parent process)

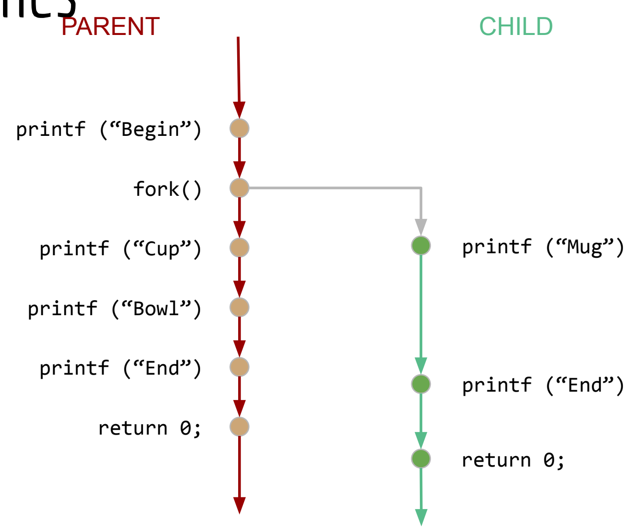
Fork()

```
/* parent */
int main() {
  printf (“Begin\n”);
  pid_t who = fork();
  if (who == 0)
     printf (“Mug %d\n, getpid());
  else {
     printf (“Cup %d\n, who);
     printf (“Bowl %d\n”, getpid());
  }

  printf (“End\n”);
  return 0;
}
```

```
/* child */
int main() {
  printf (“Begin\n”);
  pid_t who = fork();
  if (who == 0)
     printf (“Mug %d\n, getpid());
  else {
     printf (“Cup %d\n, who);
     printf (“Bowl %d\n”, getpid());
  }

  printf (“End\n”);
  return 0;
}
```

Parent & Child Time lines

```
/* parent */
int main() {
  printf (“Begin\n”);
  pid_t who = fork();
  if (who == 0)
     printf (“Mug %d\n, getpid());
  else {
     printf (“Cup %d\n, who);
     printf (“Bowl %d\n”, getpid());
  }

  printf (“End\n”);
  return 0;
}
```



exit() & wait()

- `exit(N)`: terminate and report its status number (*N*) to parent
  - Every process in Unix/Linux (except `init`) has a parent
  - exit() should be called by a “child” process
  - Automatically called when returning from main()
    - `return 71;` translates to `exit(71);`
- `wait()`: wait for a child to terminate, and accept its status
  - `wait()` should be called by a “parent” process who spawns child processes

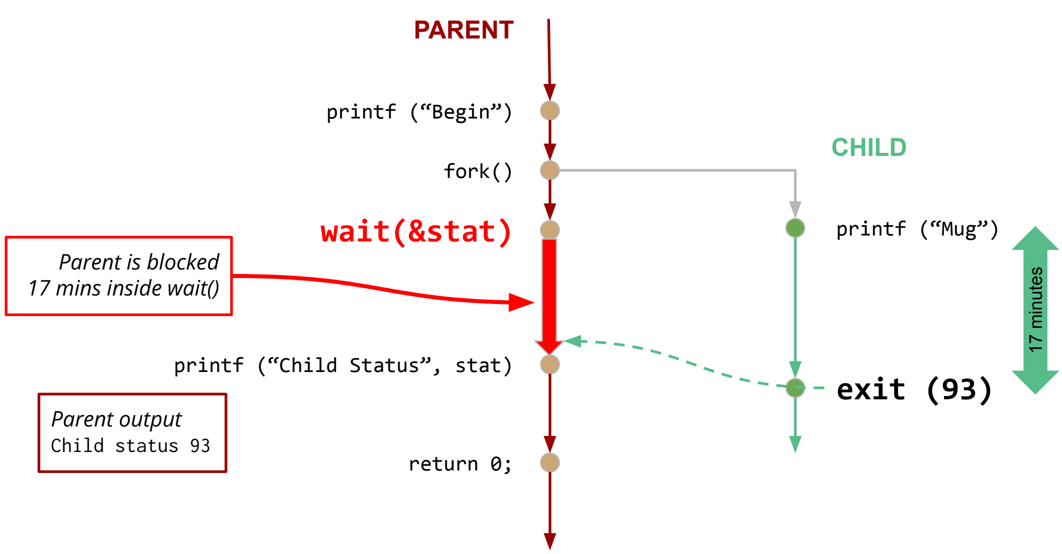
Parent-Child Handshake: `exit()` ⇔ `wait()`

```c
/* parent */
int main() {
  pid_t who = fork();
  if (who == 0) {
     printf ("Mug %d\n, getpid());
     exit (93);
  }
  else {
     int status;
     who = wait (&status);
     printf ("Child status is %d\n",
        WEXITSTATUS(status));
  }
  return 0;
}
```

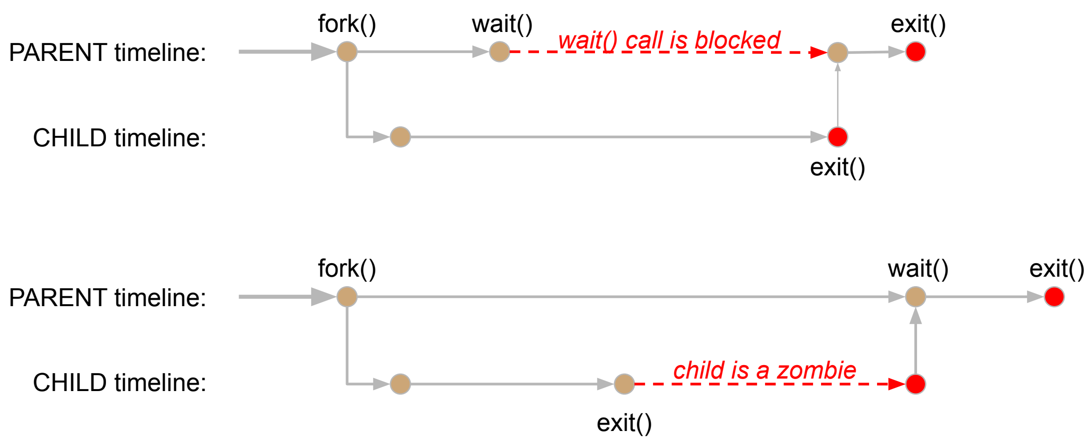
parent output "Child status is 93"

```c
/* child */
int main() {
  pid_t who = fork();
  if (who == 0) {
     printf ("Mug %d\n, getpid());
     exit (93); // any number you like
  }
  else {
     int status;
     who = wait (&status);
     printf ("Child status is %d\n",
        WEXITSTATUS(status));
  }
  return 0;
}
```

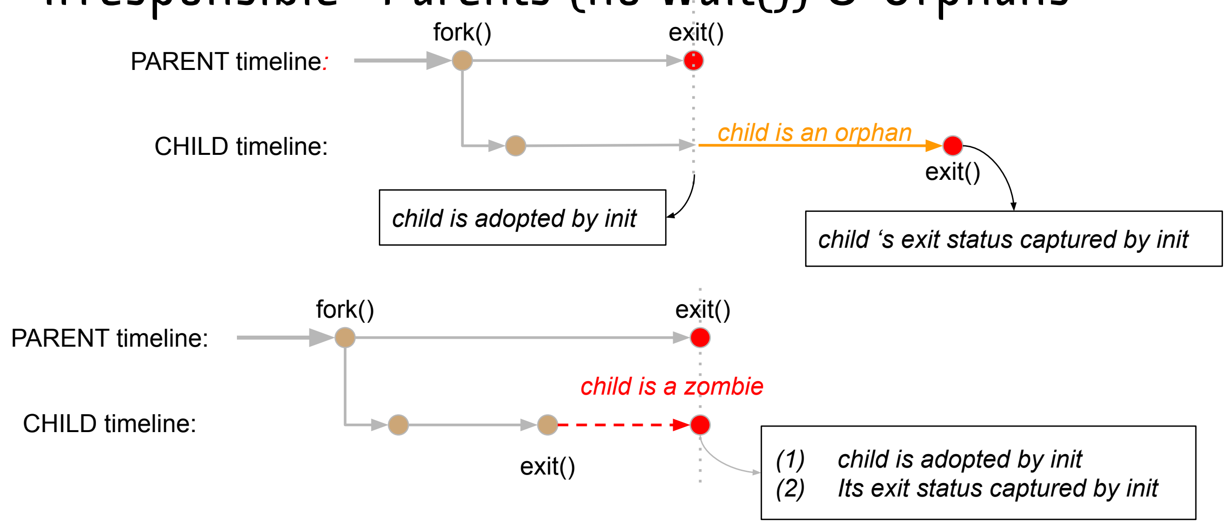

Parent & Child Time lines



Zombies (‹defunct› processes)



“Irresponsible” Parents (no wait()) & Orphans

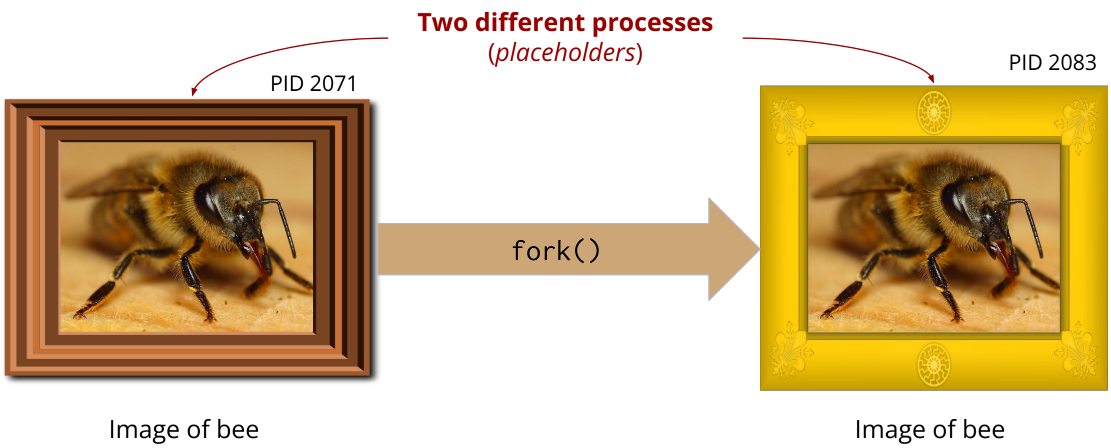


exec*(): load an **external** executable

- **Replace** the current process image with a new binary executable
  - Continue running from the "main()" of the new executable
- The new binary executable does NOT entail a new process
  - The current process is the "home" of the new binary executable
- The current process image **stays intact** if the replacement executable cannot be loaded
  - Continue running from the "next" statement in the current process image

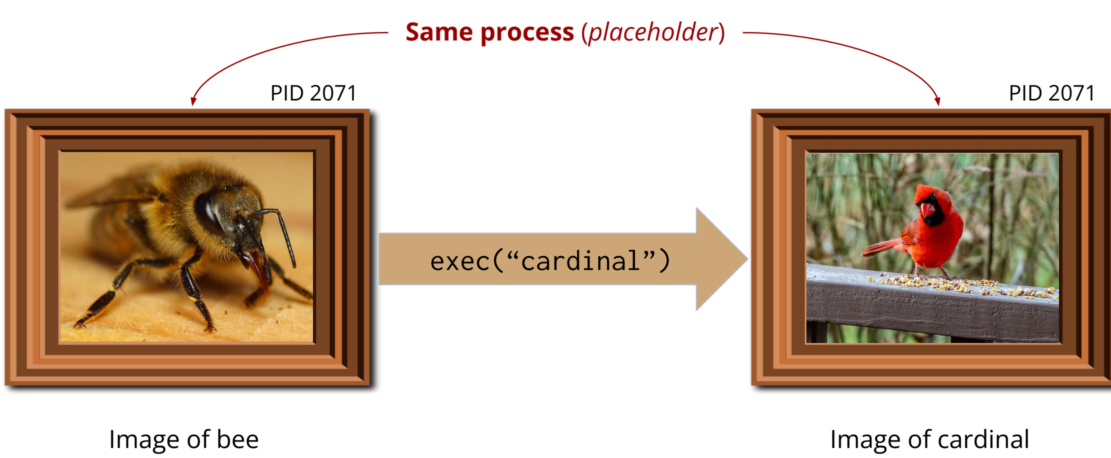
Photo Album ⇒ Frame + Image

fork()



Two different processes
(placeholders)
PID 2071
PID 2083
fork()
Image of bee
Image of bee

exec()



Same process (placeholder)
PID 2071
PID 2071
exec("cardinal")
Image of bee
Image of cardinal

Exec() failure

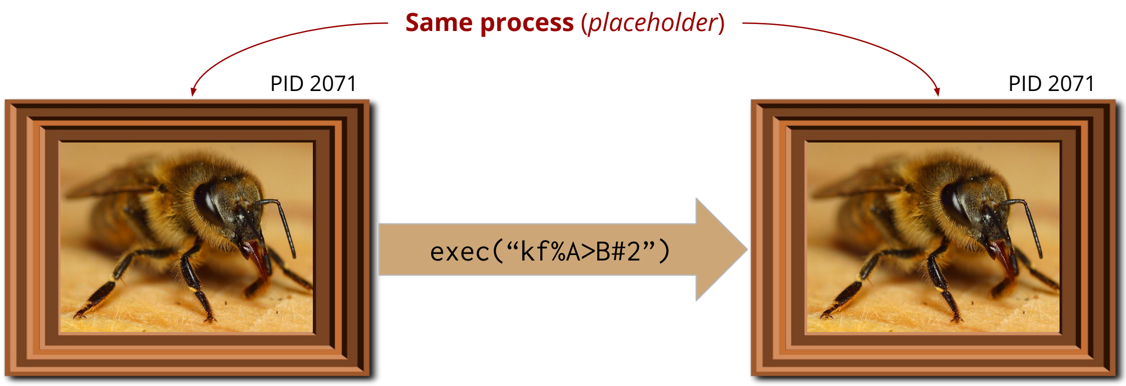


Image of bee

Image *unchanged*

`exec*()` variants

- `execl()/execlp()`: the “list” variant
  - (command line) arguments are supplied to the new binary executable using a list
- `execv()/execvp()`: the “vector” variant
  - (command line) arguments are supplied to the new binary executable using an array
- the “p” suffix: use the PATH environment variable to search for the new binary executable
- The FIRST argument to exec* is the **location** of the new binary executable
- The second (and remaining arguments) are arguments passed to the new binary executable

`exec*()` demo
`exec_cal.c` (EOS)

Using exec*()

```
int main() {
   printf ("Begin\n");
   execl ("/usr/bin/cal", "Venus", NULL);
   printf ("End\n");
   while (1) {}
   return 0;
}
```

when "`/usr/bin/cal`" **replaces** the process image

- **"End" will never get printed!**
- **the infinite while-loop will never run**

```
int main() {
   printf ("Begin\n");
   execl ("/usr/bin/california", "Venus", NULL);
   printf ("End\n");
   return 0;
}
```

"/usr/bin/california" was not found
the current process image was NOT replaced
"End" will be printed!

execl() vs. execv(): passing arguments

```
// myprog.c
void main(int argc, char*argv[])
{
  for (int k = 0; k < argc; k++)
    printf("Arg-%d: |%s|\n", argv[k]);
}
```

```
execl ("/path/to/myprog", "23", NULL);
Arg-0: |23|
```
argc: 1

```
execl ("/path/to/myprog", "23", "and me", NULL);
Arg-0: |23|
Arg-1: |and me|
```
argc: 2

```
char* args[] = {"23", NULL};
execv ("/path/to/myprog", args);
Arg-0: |23|
```
argc: 1

```
char* args[] = {"23", "and me", NULL};
execv("/path/to/myprog", args);
Arg-0: |23|
Arg-1: |and me|
```
argc: 2


```
ls  -l -a -R
```

```
execl("/usr/bin/ls", "GVSU", "-l", "-a", "-R", NULL);
```

```
execlp("ls", "GVSU", "-l", "-a", "-R", NULL);
```

```
char* arr[] = {"-l", "-a", "-R", NULL};
execv("/usr/bin/ls", arr);
```

```
char* arr[] = {"-l", "-a", "-R", NULL};
execvp("ls", arr);
```

Time To Disclose the Truth

- **False statement**: Parent and child processes DO NOT share any contents of code/data/heap/stack
- Facts
  - Code section of both parent and child are exact copy of each other (**shareable**)
  - Heap section are very likely to be exact copy of each other (**shareable**)
  - Data (or stack) section may differ in a few bytes (**when the return value of fork() is saved to a global (or local) variable**)

COW (Copy-on-Write)

- Code section (always read-only) can be shared
- Data, Heap, and Stack sections of parent and child can be shared IF these sections are used in **read-only** fashion
- If either parent/child **attempts to modify/write** data, heap, or stack, must be allocated **its own copy** (the OS does it for you)
- *Class discussion*: how to enforce COW?

Program Execution

```
int main() {  // Program A

  while (1);
  return 0;
}
```

- Is program A **running on the CPU** at all time?
  - Why?
  - Why not?

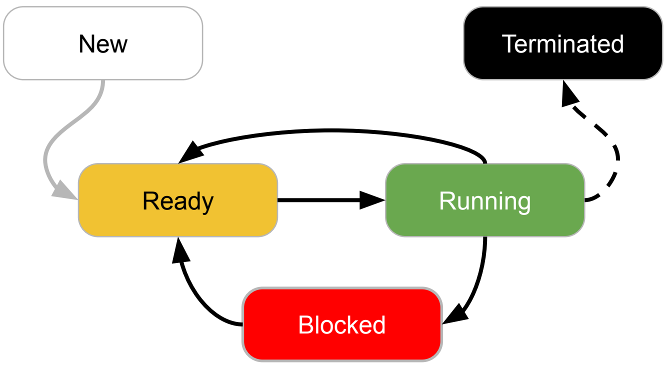
```
int main() {  // Program B
  int num;
  while (1) {
    printf (“Number? “);
    scanf (“%d”, &num);
    printf (“[%d]\n”, num);
  }
  return 0;
}
```

- Is program B **running on the CPU** at all time?
  - Why?
  - Why not?
- How is it different from the execution of Program A?

Sharing (Virtualize) the CPU?

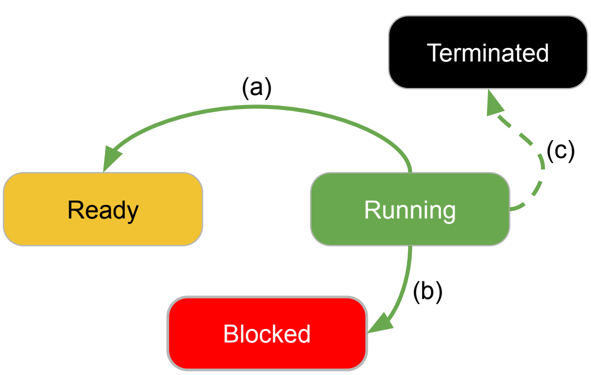
Create N virtual CPUs out of ONE physical CPU?

Process State Transition Diagram



- New → Rdy: the process just created, ready to use the CPU
- Rdy → Run: dispatched by OS to use the CPU
- Run → Term: the process exits
- Run → Rdy: process time slice expired
- Run → Blk: the process issued a blocking system call (read(), sleep(), wait() for child, wait for mouse click, ….
- Blk → Rdy: the blocking syscall completed, the process is ready to use the CPU again

Process "Eviction" from CPU

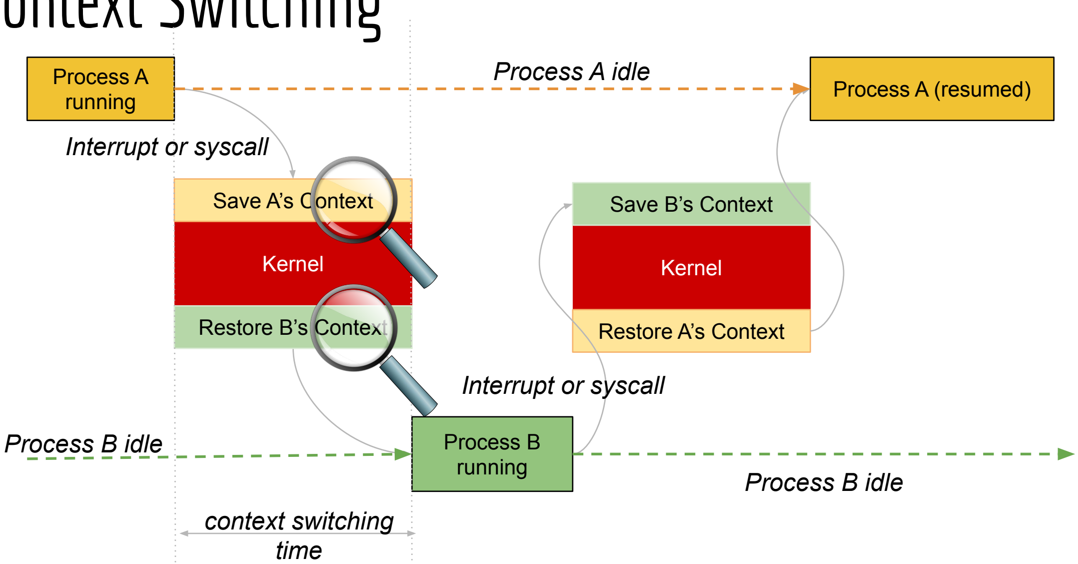


- The current process is evicted from CPU
  - The CPU becomes vacant
- What triggers the eviction?
- What is the nature of eviction?
  - Abrupt or gradual?
- For (a) and (b), what needs to be done to **properly resume** an evicted process?

How to share the CPU?

- When a program is **running**, its data can be found in
  - RAM (data section, heap section, stack section)
  - and also in ___________
- What are potential problems in sharing the CPU?
- Solution?
- How do we share (and avoid conflict):
  - A classroom in a campus building?
  - A CPU?

Context Switching



```
Save Program Counter to A's stack
//
// Other assembly instructions here
//
```

Save **CPU state** of A

SysCall Work by Kernel

```
//
// Other assembly instructions here
//
Load Program Counter with address from B's stack
```

Restore B's **CPU state**

Process Control Block (PCB)

- PCB is an OS data structure for storing important information of a process
  - Current context: CPU registers, Program Counter, control registers, ...
  - Scheduling-related information: priority, wait time, total CPU time, ...
  - Memory-related info: total memory, segment size, page tables,
  - I/O related info
  - Accounting info: CPU time used, memory used (real mem, virtual mem), ...

Process Mgmt ⇔ Processor Mgmt

- Process(or) Scheduler: dispatch a process to the CPU from the ready Q
- Queues for managing processes
  - Ready Queue: keeps all the processes who are in Ready state
  - Device Queue(s) or I/O Queue(s): keeps all the processes who are in Blocked state, waiting for I/O completion to/from a particular device
- Schedulers
  - Short-Term or CPU Scheduler: dispatches a process from the Ready Q to use the CPU
  - Medium-Term Scheduler: decides which process gets swapped-in / swapped-out (between RAM and swap disk)
  - Long-Term Scheduler (in a batch system): dispatch batch jobs to enter the system

Schedulers