System Programming: Process Management

Raymond Namyst

Dept. of Computer Science

University of Bordeaux, France

https://gforgeron.gitlab.io/progsys/

Processes

- Processes are lively instances of programs
  - Program = binary code stored on disk
  - Multiple processes can run the same program independently

Processes

- Processes are lively instances of programs
  - Program = binary code stored on disk
  - Multiple processes can run the same program independently

- Process = Address Space + Execution Context
  - Address space

Processes

- Processes are lively instances of programs
  - Program = binary code stored on disk
  - Multiple processes can run the same program independently
- Process = Address Space + Execution Context
  - Address space
    - Set of visible memory addresses
      - Code, Data, Heap, Stack, Shared Libraries, etc.
  - Execution Context

Processes

- Processes are lively instances of programs
  - Program = binary code stored on disk
  - Multiple processes can run the same program independently

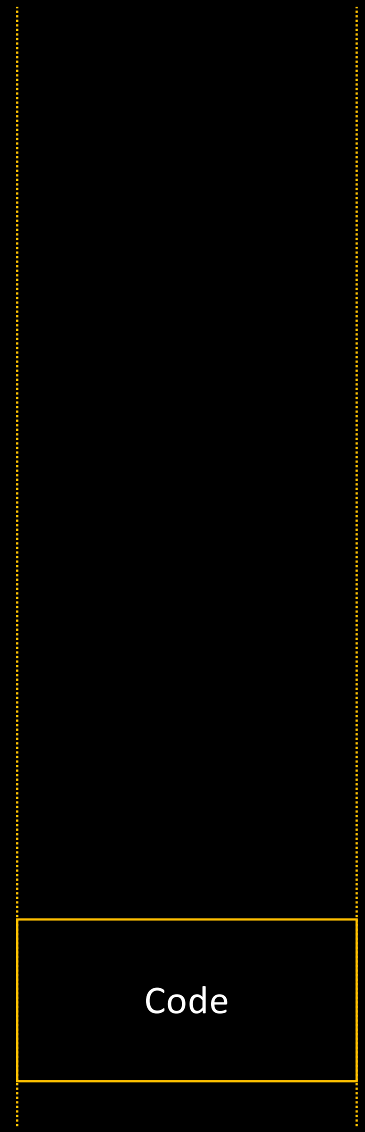
- Process = Address Space + Execution Context
  - Address space
    - Set of visible memory addresses
      - Code, Data, Heap, Stack, Shared Libraries, etc.
  - Execution Context
    - Stack + content of processor registers

Address Space

- Typically composed of non-contiguous memory regions
  - A region being a contiguous range of valid addresses

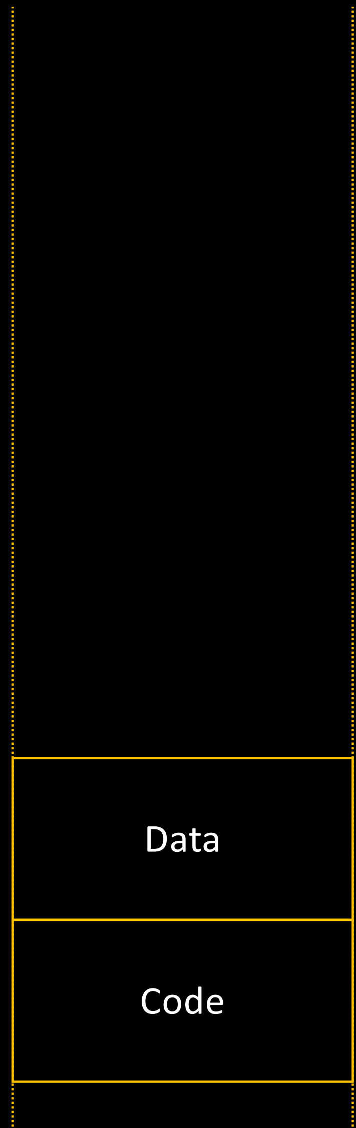
Address Space

- Typically composed of the following regions
  - Code
    - (aka text segment)
    - Contains executable instructions
    - Usually a read-only region



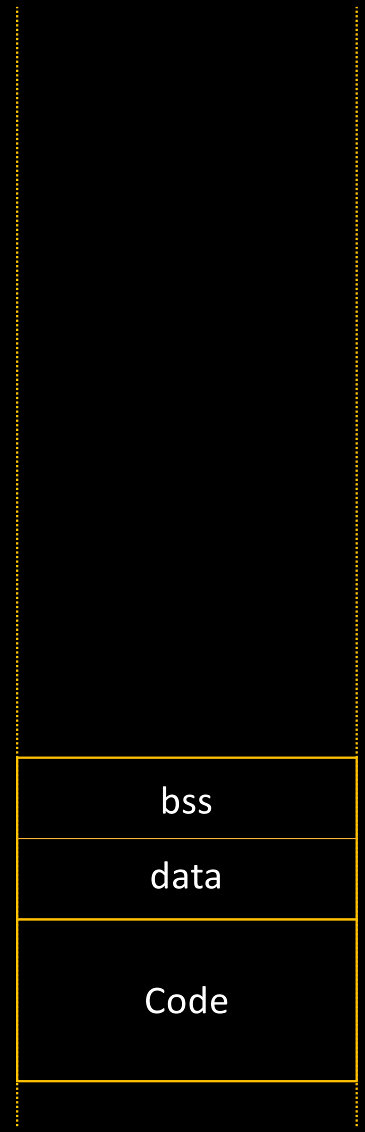
Address Space

- Typically composed of the following regions
  - Code
  - Data
    - Allocation of static variables
      - `int i;`



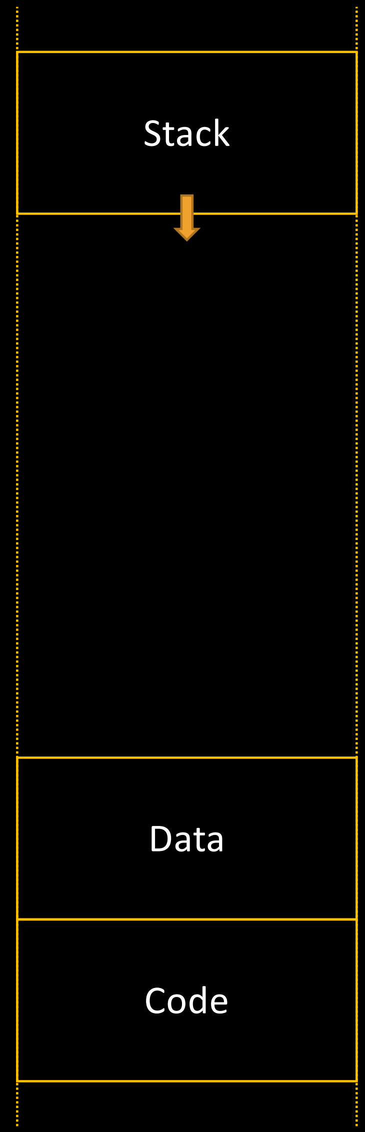
Address Space

- Typically composed of the following regions
  - Code
  - Data
    - Allocation of static variables
    - Actually two segments
      - Initialized data (data segment)
        - `float pi = 3.1415;`
        - Stored in object file
      - Uninitialized data (bss segment)
        - `int i;`
        - Only segment size is stored in object file



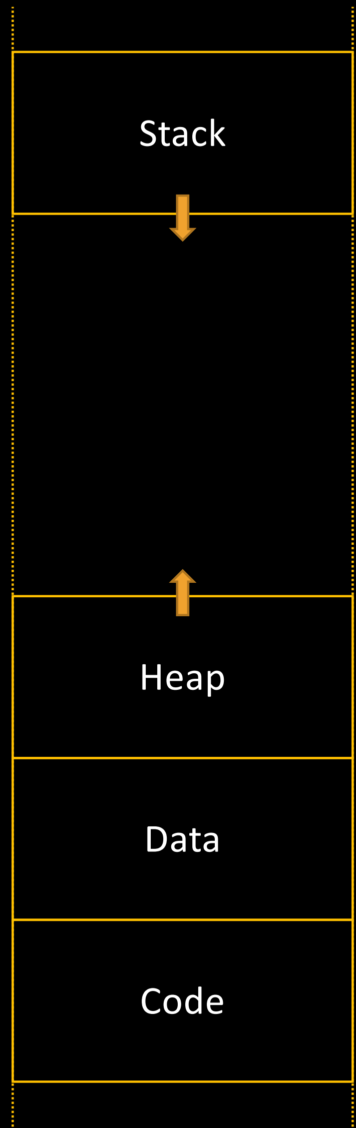
Address Space

- Typically composed of the following regions
  - Code
  - Data
  - Stack
    - Allocation of function parameters and local variables
    - Automatic growth
    - 8 MiB default limit under Linux



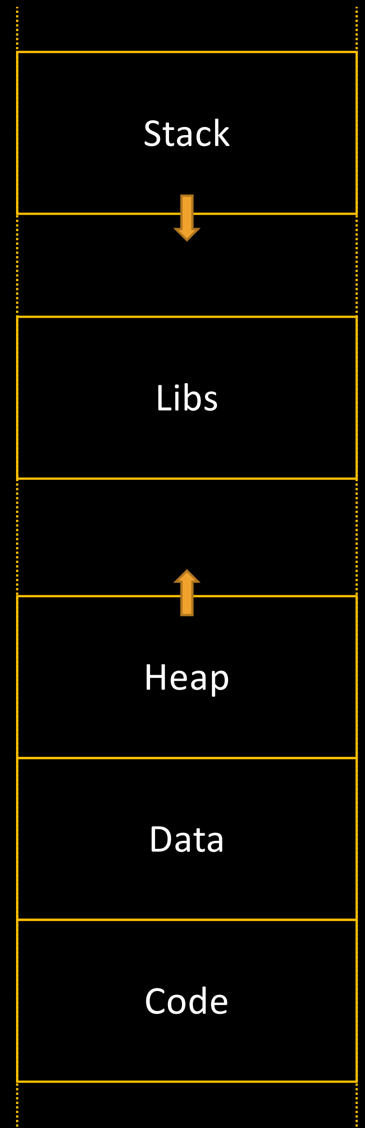
Address Space

- Typically composed of the following regions
  - Code
  - Data
  - Stack
  - Heap
    - Dynamic allocations
      - `malloc/free`
    - Managed by libc
      - Dynamic expansion
      - Note: OS cannot always detect accesses outside malloc'ed buffers…



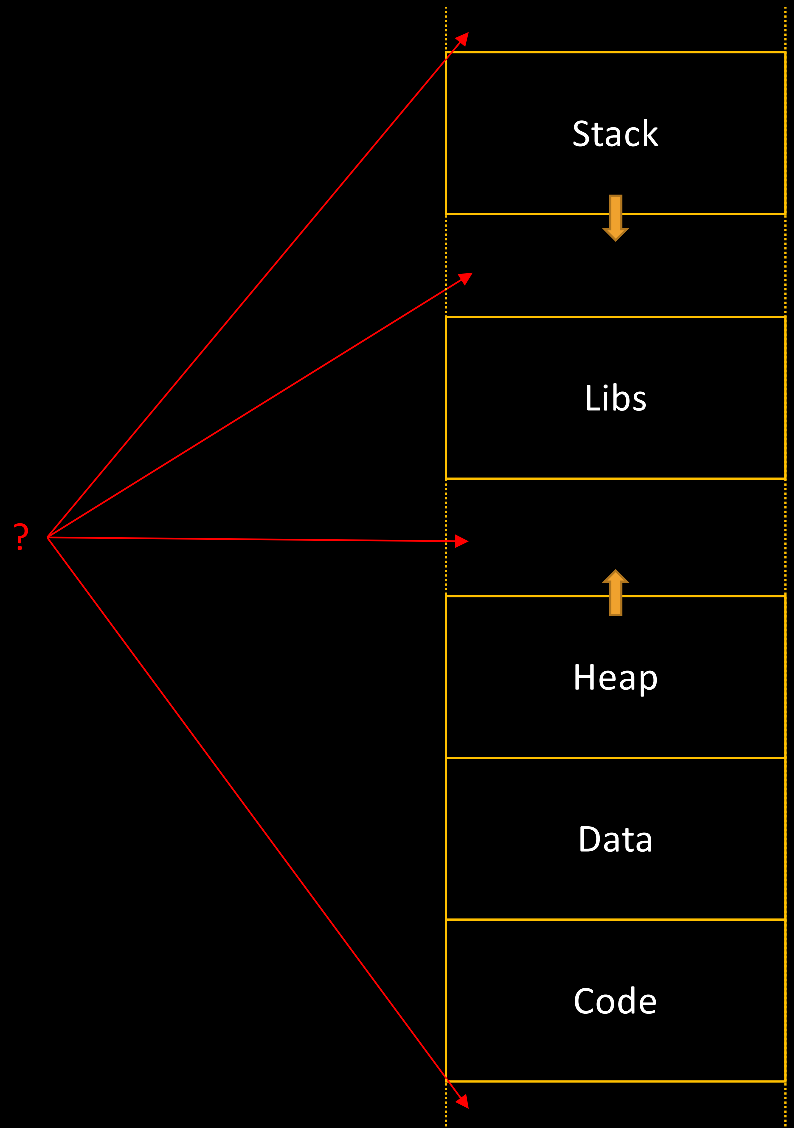
Address Space

- Typically composed of the following regions
  - Code
  - Data
  - Stack
  - Heap
  - Shared Libraries
    - libc, libm, libGL, etc.
    - Mapped on demand



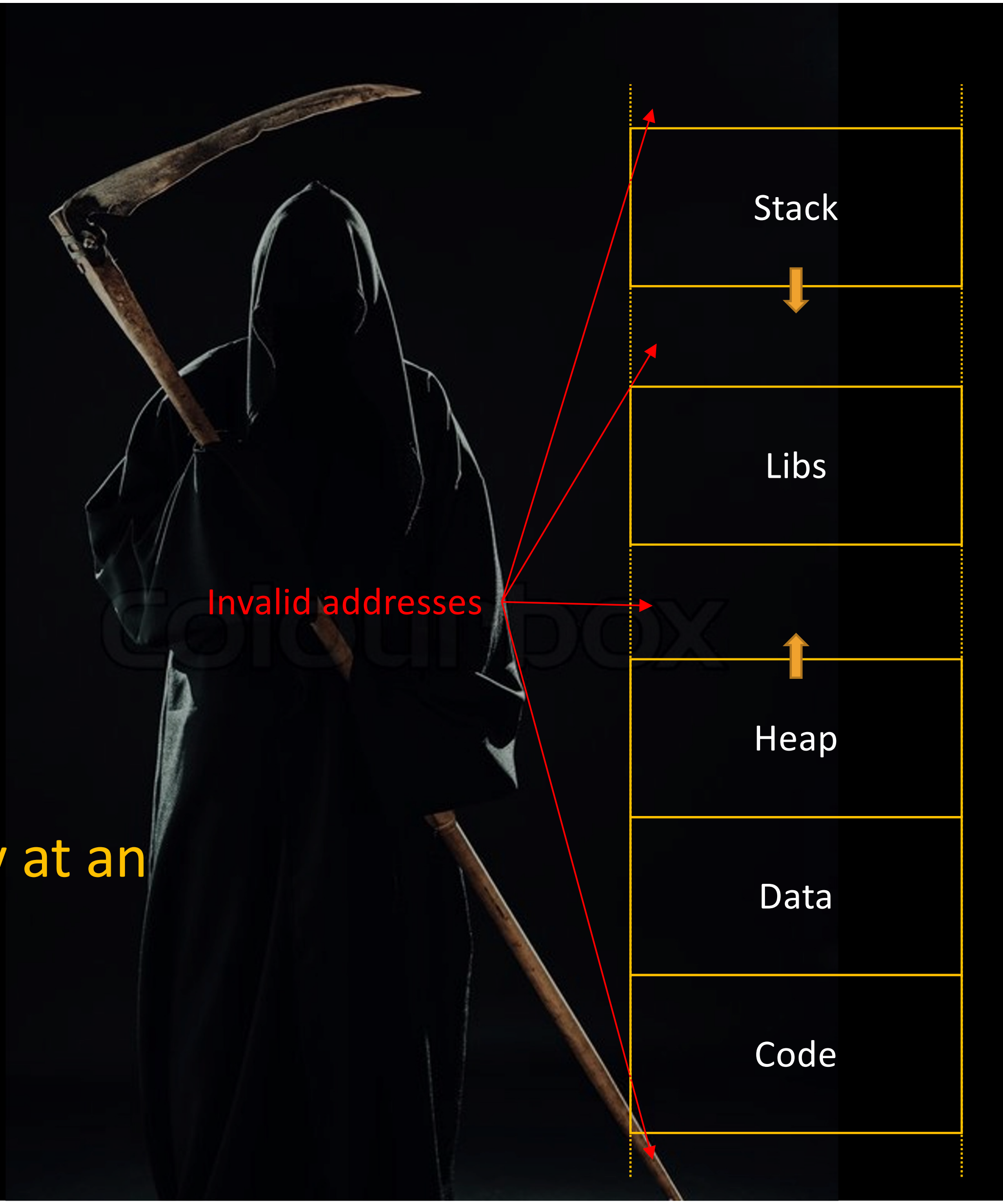
Address Space

- Typically composed of the following regions
  - Code
  - Data
  - Stack
  - Heap
  - Shared Libraries
- What do these placeholder contain?



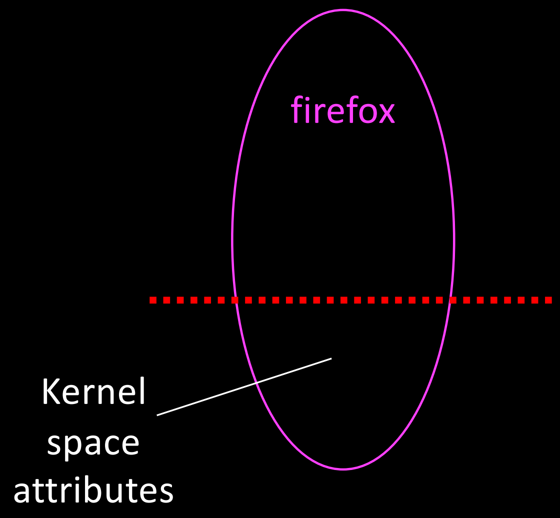
Address Space

- Typically composed of the following regions
  - Code
  - Data
  - Stack
  - Heap
  - Shared Libraries
- Attempt to access memory at an invalid address leads to a Segmentation Fault



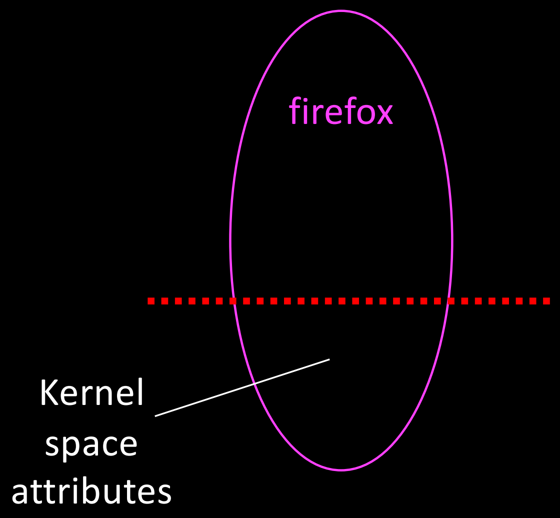
Process Context

- In addition to Address Space description, the kernel stores the following information about each process:
  - Process ID (pid) -> see `getpid()`
  - Priority
  - User ID (real/effective)
  - File descriptor table
  - Space for registers backup
  - Etc.

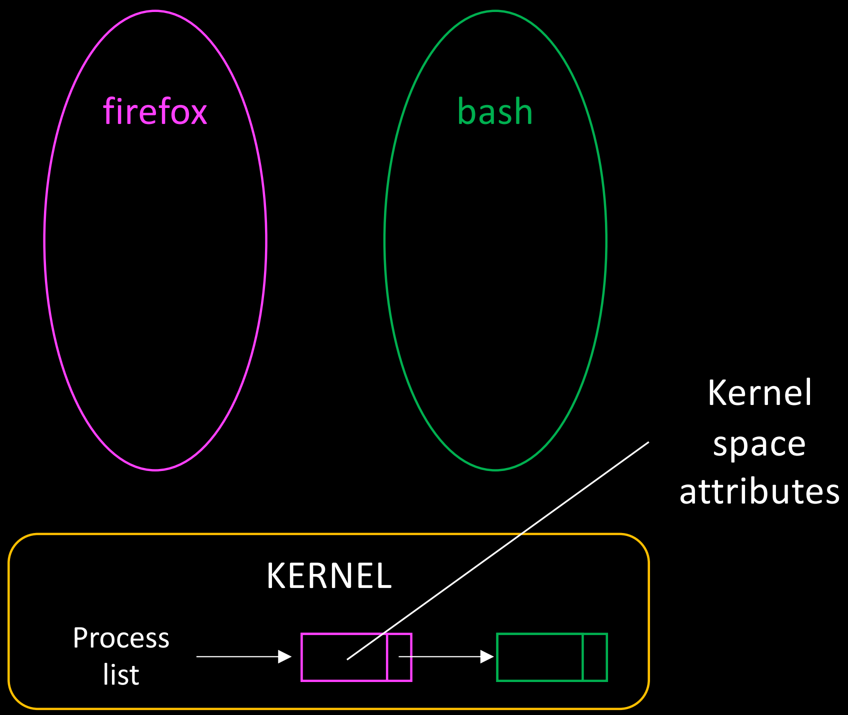


Reminder about process representation

Processes can be
represented this way:



But reality is (obviously) more like:



Process Identity

```
int main(int argc, char *argv[])
{
  printf (“Hello from %d\n”,
          getpid());
}
```

```
[mymachine] ./getpid
Hello from 38043
[mymachine]
```

Process Identity

```
int main(int argc, char *argv[])
{
  printf ("Hello from %d\n",
          getpid());
}
```

```
[mymachine] ./getpid
Hello from 38043
[mymachine] ./getpid &
[1] 38044
[mymachine] Hello from 38044
```

Process Creation

- One system call
  - `pid_t fork ();`
- Fork clones the calling process
  - The whole address space is copied
  - Right after fork, father & child see the same values
    - But they don't share any memory
- Fork returns
  - On father's side: the pid of the newborn process
  - On child's side: 0

Process Creation

```
int main(int argc, char *argv[])
{
  fork ();
  printf (“Hello from %d\n”,
          getpid());
}


// see first-fork.c
```

Process Creation

```
int main(int argc, char *argv[])
{
  fork ();
  printf ("Hello from %d\n",
          getpid());
}


// see first-fork.c
```

```
[mymachine] ./fork
Hello from 33440
Hello from 33441
[mymachine]
```

Process Creation

```
int main(int argc, char *argv[])
{
  fork (); fork();
  printf ("Hello from %d\n",
          getpid());
}
```

Process Creation

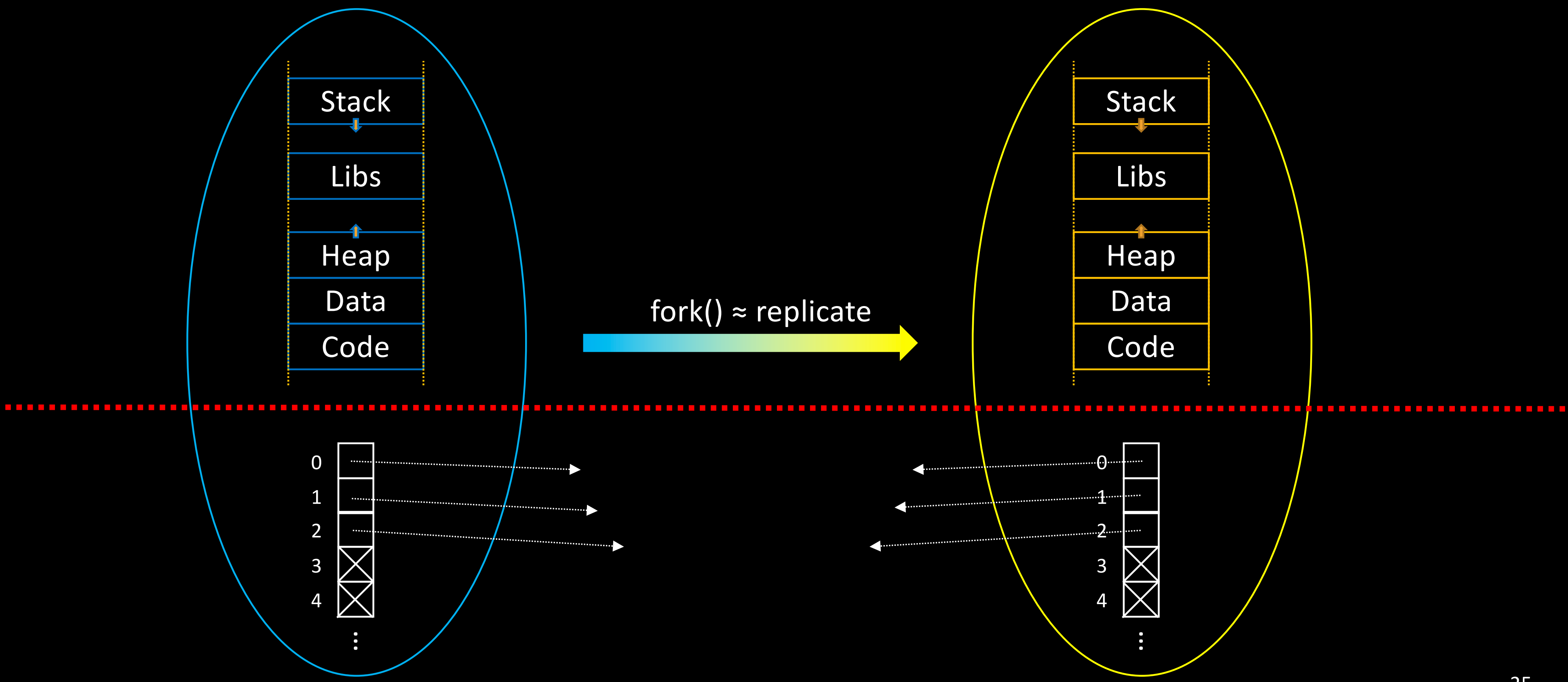
```
int main(int argc, char *argv[])
{
  fork (); fork();
  printf (“Hello from %d\n”,
          getpid());
}
```

```
[mymachine] ./fork
Hello from 33463
Hello from 33465
Hello from 33464
[mymachine] Hello from 33466
```

Process Creation

- Return value: fork.c

Process Creation:
fork() replicates the whole bubble!



Process Creation

- Global variables: vars-n-fork.c

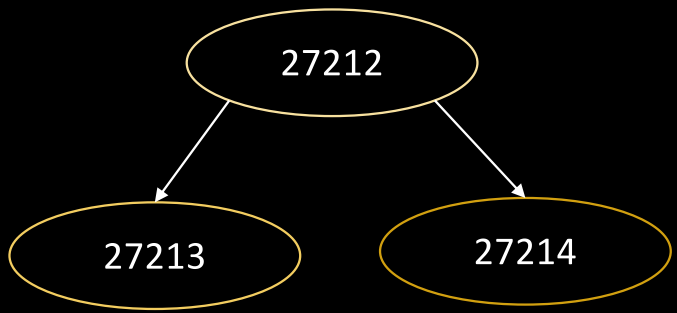
Multiple fork() calls

```c
int main (int argc, char *argv[])
{
  pid_t pid[2];

  pid[0] = fork ();
  if (pid[0]) { // father
    pprintf ("Parent's fork return value: %d\n", pid[0]);
    pid[1] = fork ();
    if (pid[1]) // father
      pprintf ("Parent's fork return value: %d\n", pid[1]);
    else // Child
      pprintf ("Child's fork return value: %d\n", pid[1]);
  } else // Child
    pprintf ("Child's fork return value: %d\n", pid[0]);

  return 0;
}
```

Multiple fork() calls

```
int main (int argc, char *argv[])
{
  pid_t pid[2];

  pid[0] = fork ();
  if (pid[0]) { // father
    pprintf ("Parent's fork return value: %d\n", pid[0]);
    pid[1] = fork ();
    if (pid[1]) // father
      pprintf ("Parent's fork return value: %d\n", pid[1]);
    else // Child
      pprintf ("Child's fork return value: %d\n", pid[1]);
  } else // Child
    pprintf ("Child's fork return value: %d\n", pid[0]);

  return 0;
}
```

```
[mymachine] ./forkfork
[PID 27212] Parent's fork return value: 27213
[PID 27212] Parent's fork return value: 27214
[PID 27213] Child's fork return value: 0
[PID 27214] Child's fork return value: 0
```



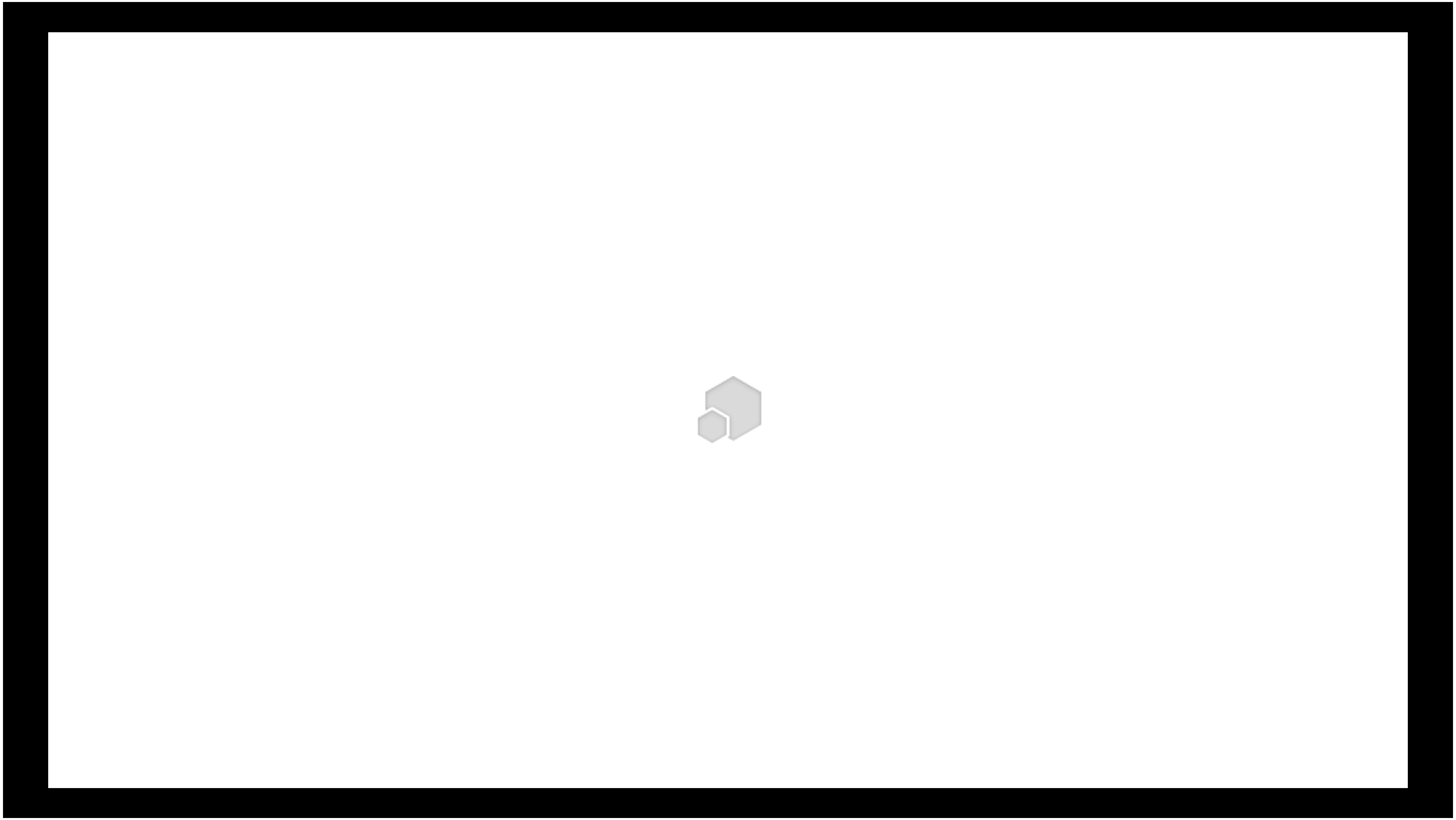
Multiple fork() calls

```
int main (int argc, char *argv[])
{
  pid_t pid[2];

  pid[0] = fork ();
  if (pid[0]) { // father
    pprintf ("Parent's fork return value: %d\n", pid[0]);
  } else { // Child
    pprintf ("Child's fork return value: %d\n", pid[0]);
    pid[1] = fork ();
    if (pid[1]) // father
      pprintf ("Parent's fork return value: %d\n", pid[1]);
    else // Child
      pprintf ("Child's fork return value: %d\n", pid[1]);
  }
  return 0;
}
```

Multiple fork() calls

```
int main (int argc, char *argv[])
{
  pid_t pid[2];

  pid[0] = fork ();
  if (pid[0]) { // father
    pprintf ("Parent's fork return value: %d\n", pid[0]);
  } else { // Child
    pprintf ("Child's fork return value: %d\n", pid[0]);
    pid[1] = fork ();
    if (pid[1]) // father
      pprintf ("Parent's fork return value: %d\n", pid[1]);
    else // Child
      pprintf ("Child's fork return value: %d\n", pid[1]);
  }
  return 0;
}
```

```
[mymachine] ./forkfork
[PID 27588] Parent's fork return value: 27589
[PID 27589] Child's fork return value: 0
[PID 27589] Parent's fork return value: 27590
[mymachine] [PID 27590] Child's fork return value: 0
```

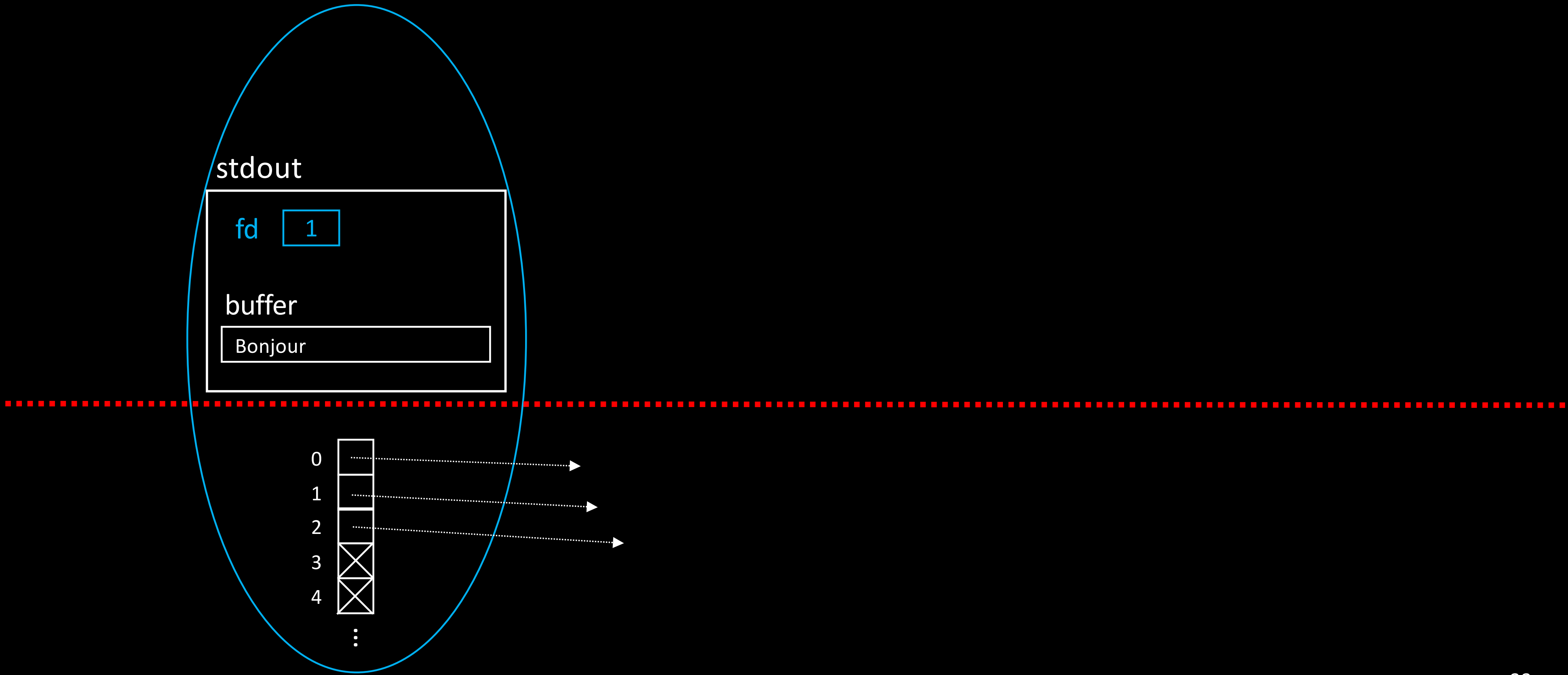


Process Creation:
fork() replicates the whole bubble!

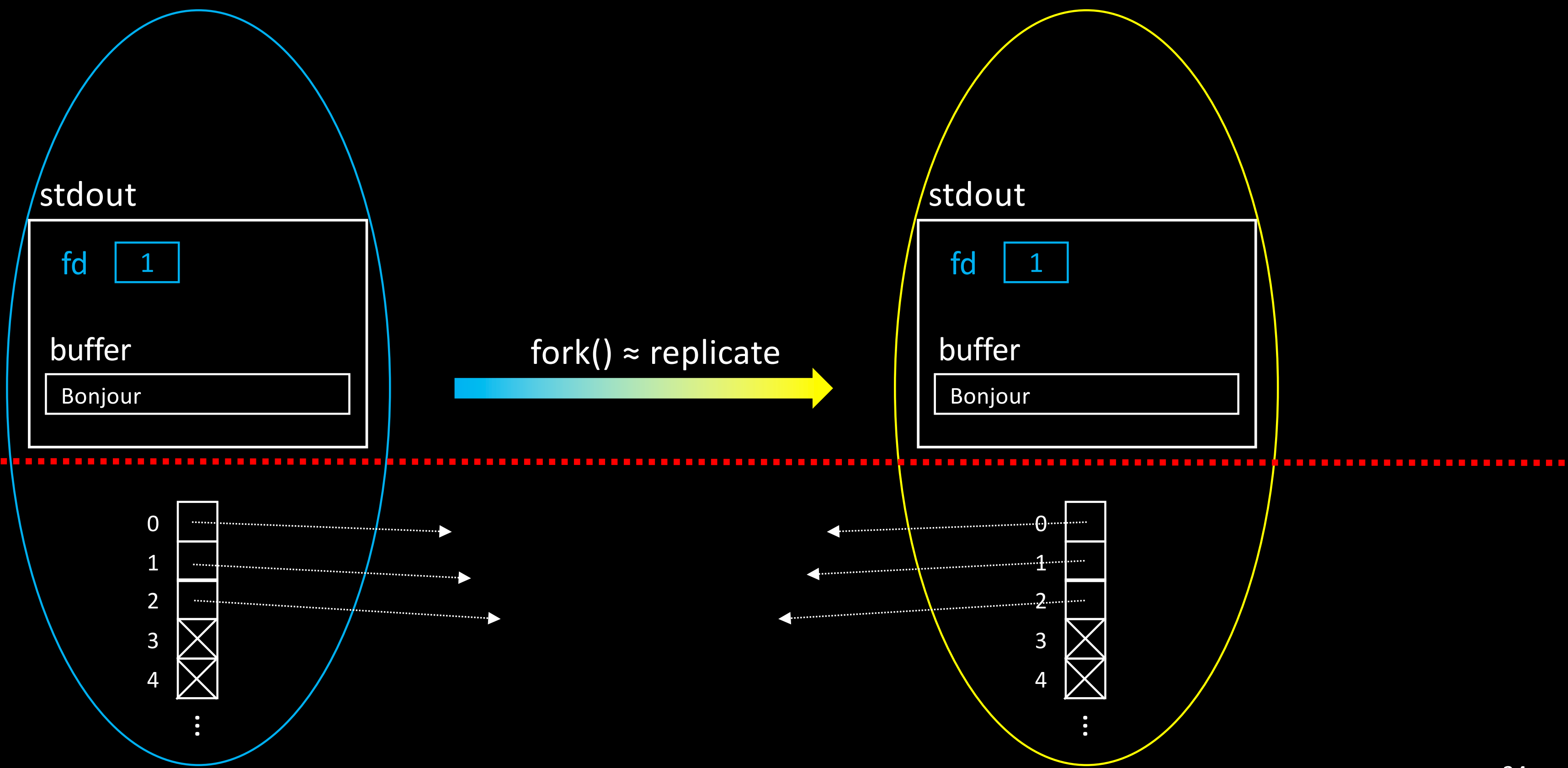
- The following program does not produce the output you might expect...

```
int main (int argc, char *argv[])
{
  printf ("Bonjour");
  fork ();

  return 0;
}
```

Process Creation:
fork() replicates the whole bubble!



Process Creation:
fork() replicates the whole bubble!

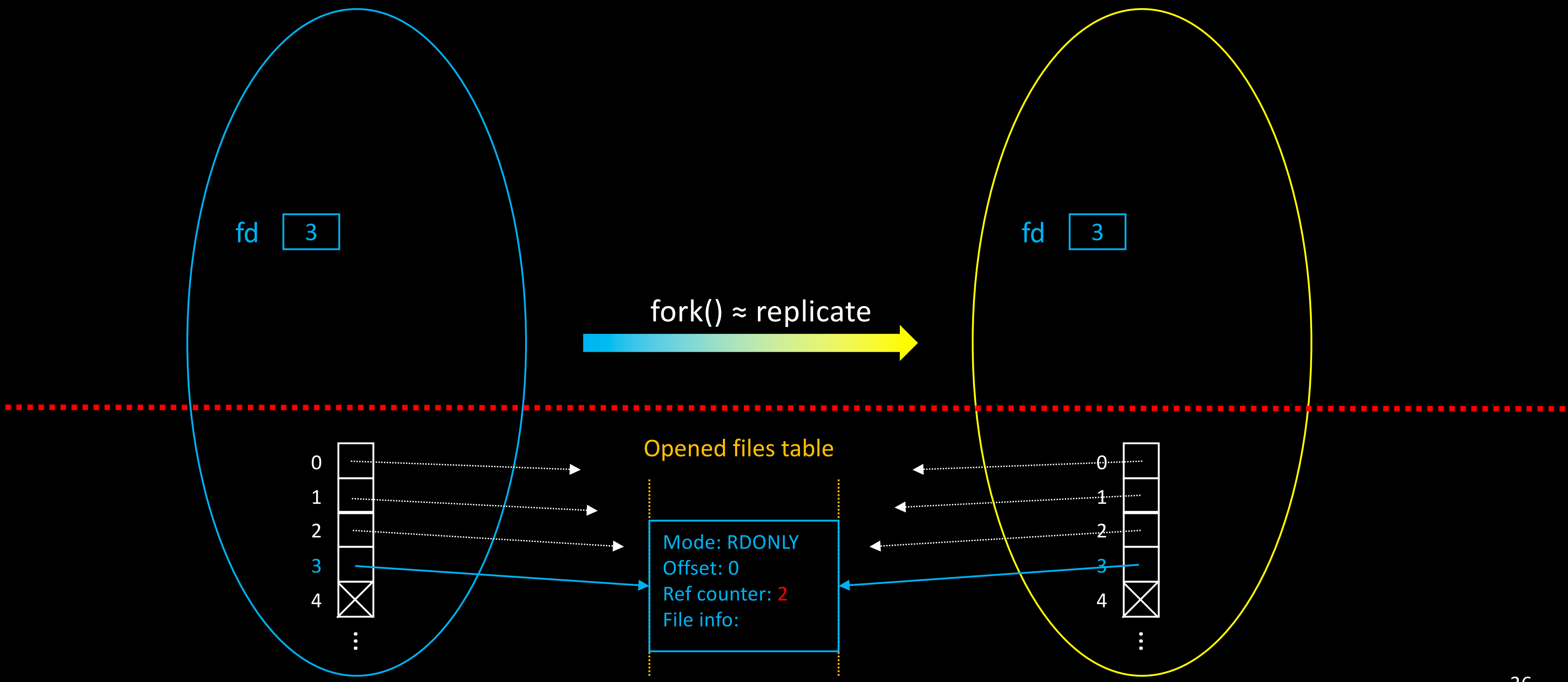


Process Creation: fork() replicates the whole bubble!

- File descriptors are not closed
  - And they share records in the opened file table
- Processes share the same file offset!

```
int main (int argc, char *argv[])
{
  int fd = open (FILENAME, O_RDONLY);
  check (fd, "Cannot open %s file",
         FILENAME);

  fork ();

  lire (fd); // See lecture.c

  close (fd);

  return 0;
}
```

Process Creation:
fork() replicates the whole bubble!



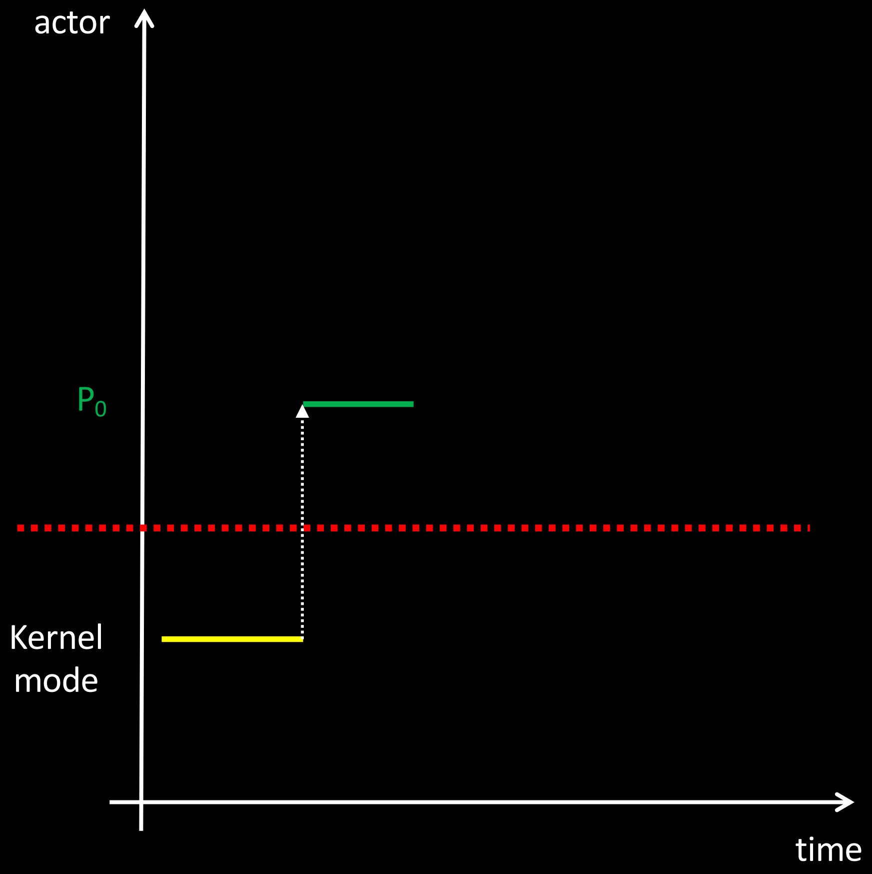
Waiting for child termination

- Wait until one child is terminated:
  - `pid_t wait (int *stat_loc);`
    - Information about termination is stored in *stat_loc
      - `WEXITSTATUS (*stat_loc)` gives return value of child process
- More powerful version:
  - `pid_t waitpid (pid_t pid, int *stat_loc, int options);`
    - pid can be -1 (= ANY)
    - options can be `WNOHANG` (= just check without blocking)

Waiting for child termination

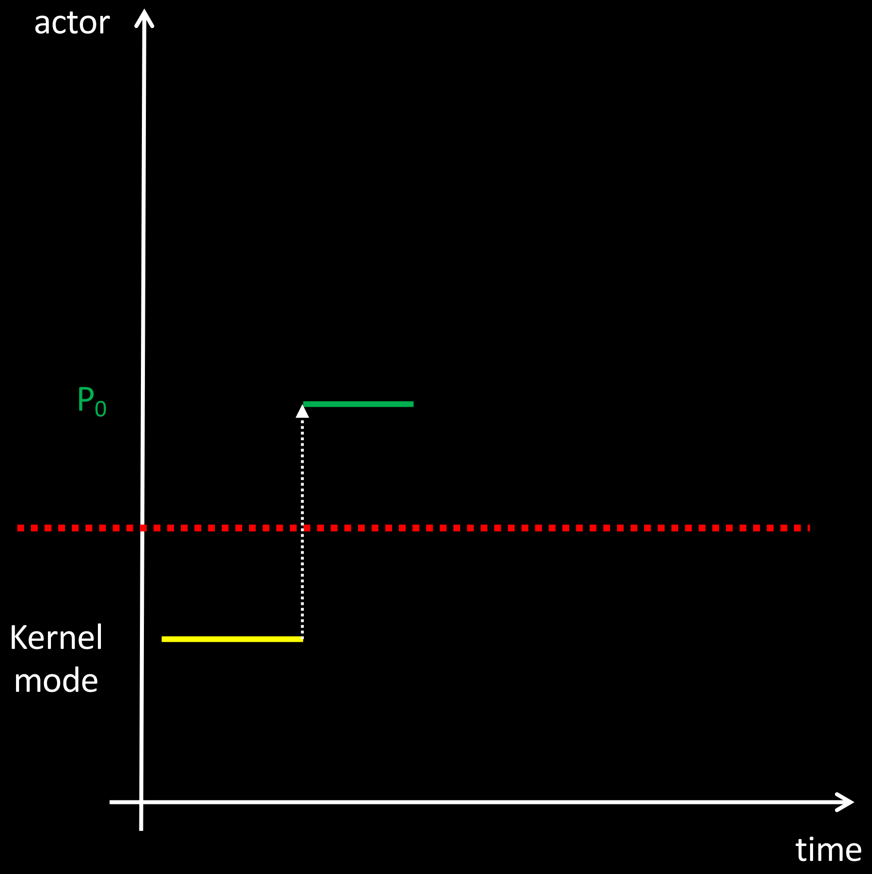
```
int main (int argc, char *argv[])
{
  pid_t pid;
  pid = fork ();
  if (pid) { // father
    int status;
    pprintf ("Parent's fork return value: %d\n", pid);
    wait (&status);
    pprintf ("Child termination detected (return code: %d)\n", WEXITSTATUS (status));
  } else { // Child
    pprintf ("Child's fork return value: %d\n", pid);
    sleep(3);
    pprintf ("Child is terminating\n");
    return 31;
  }
  return 0;
}
```

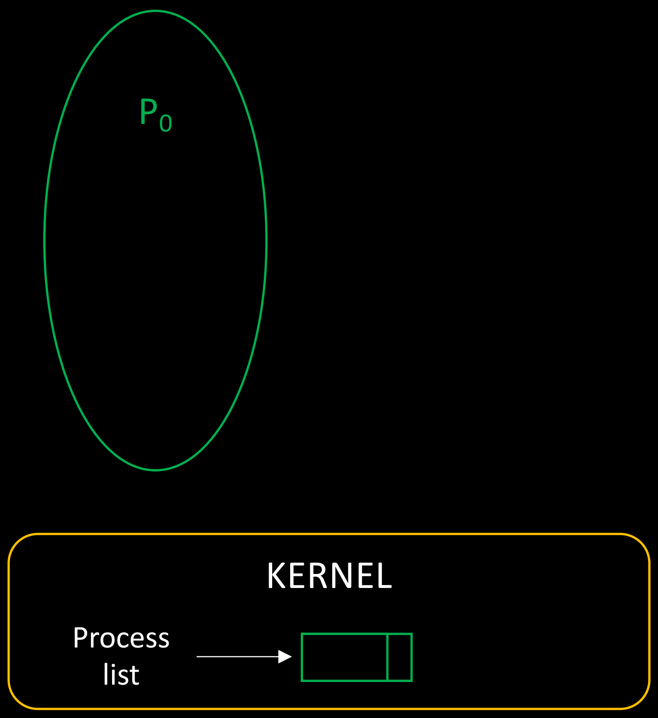
Process Creation



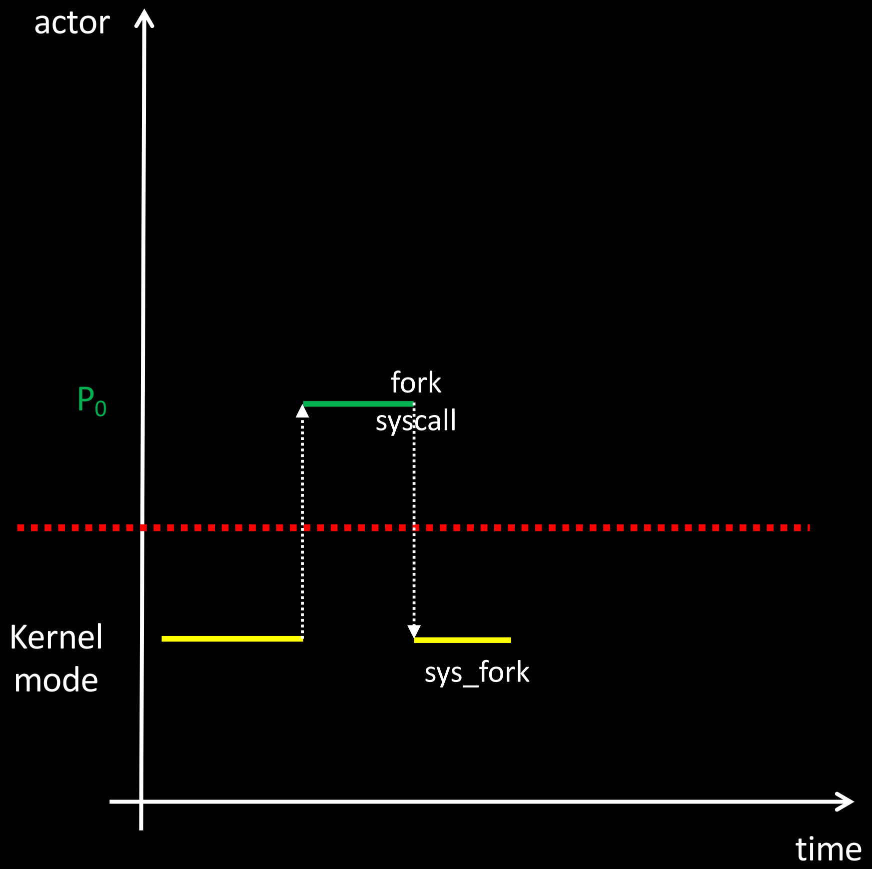
- The Kernel originally spawns one process
  - This process will in turn create several processes (background DAEMONs)
    - Using the fork() system call

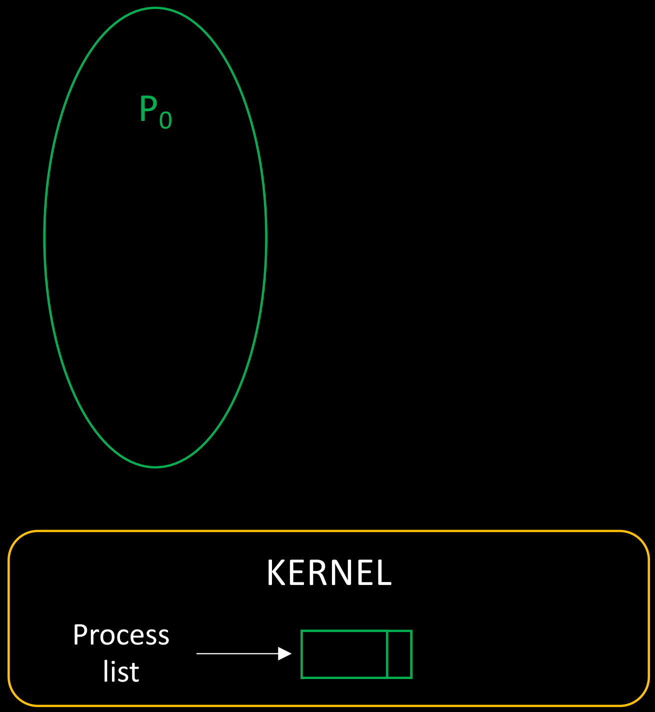
Process Creation



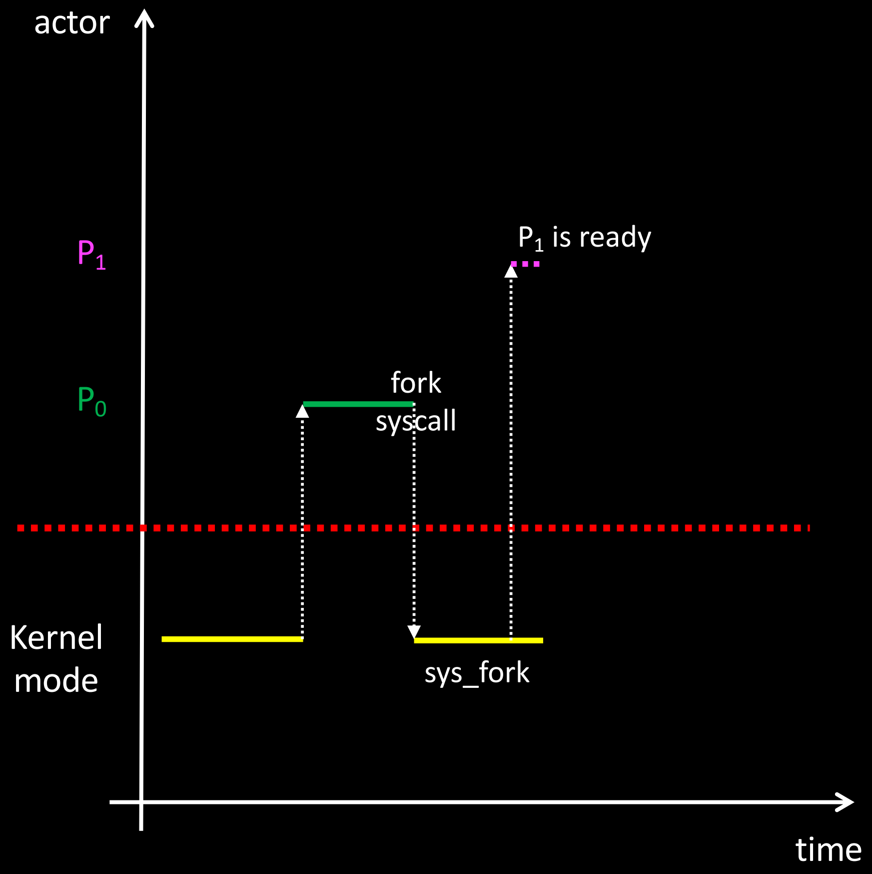


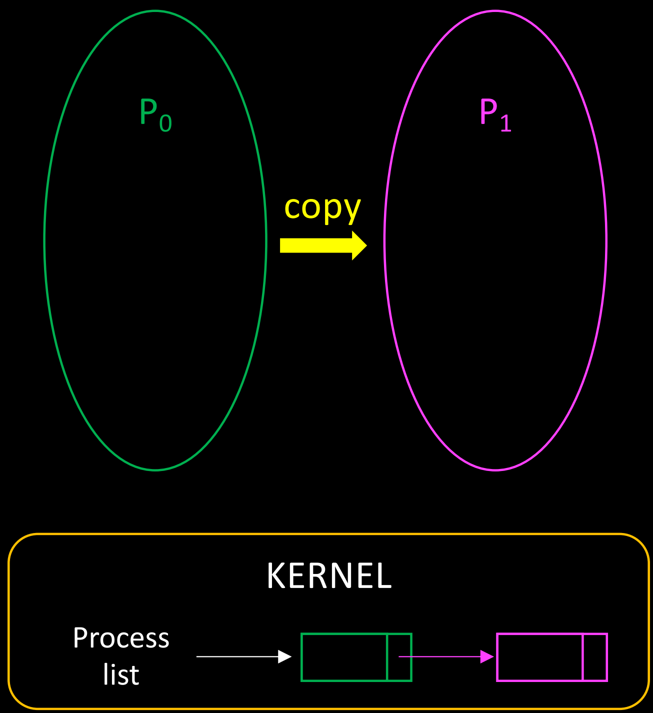
Process Creation



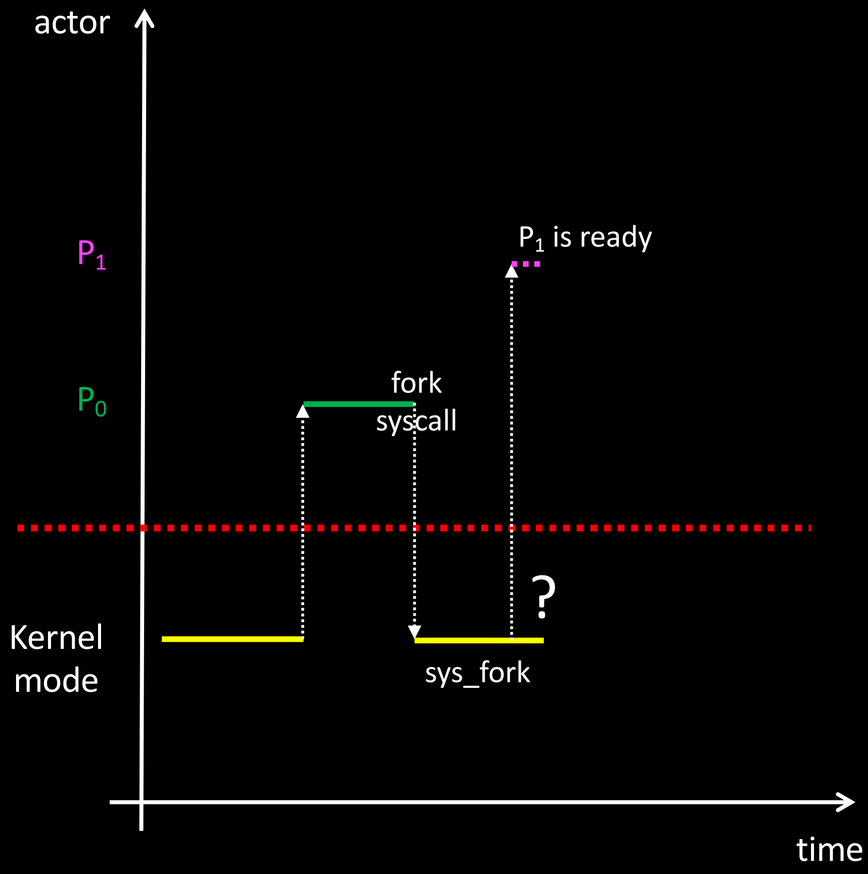


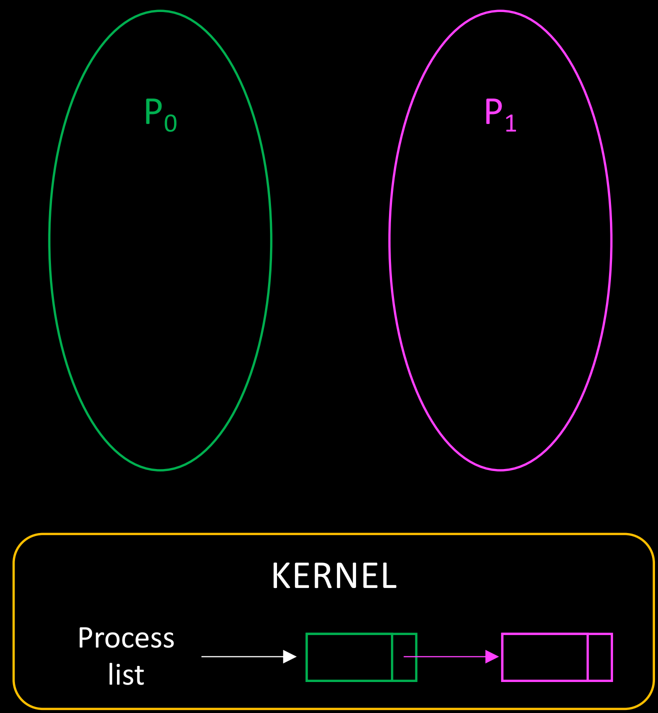
Process Creation



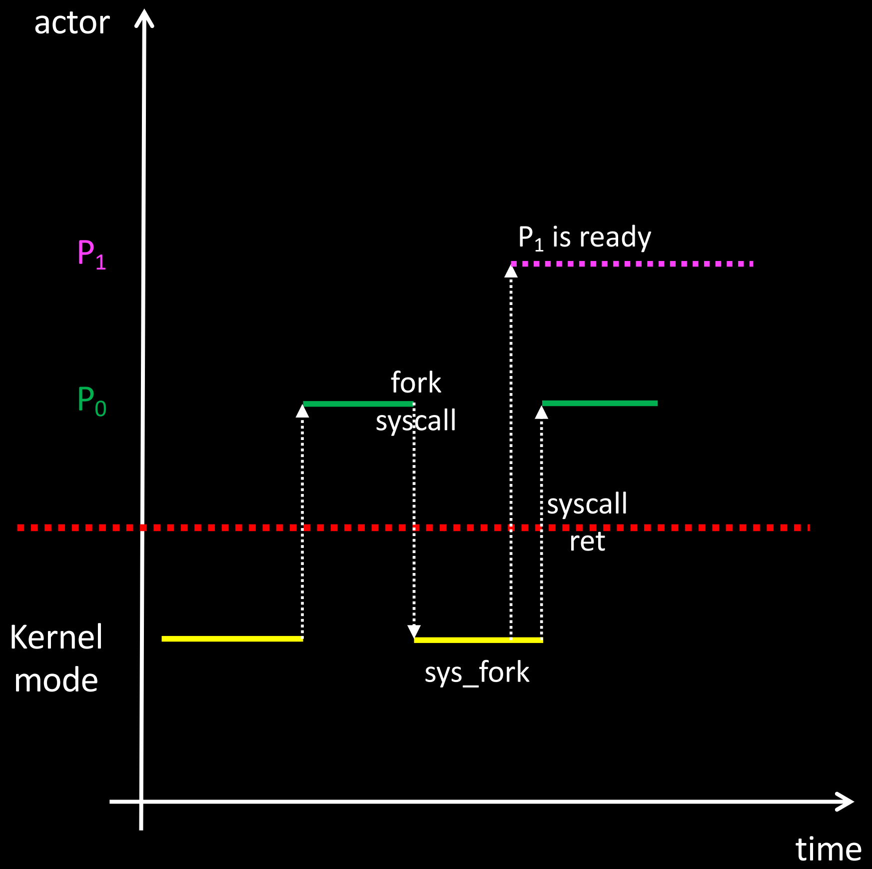


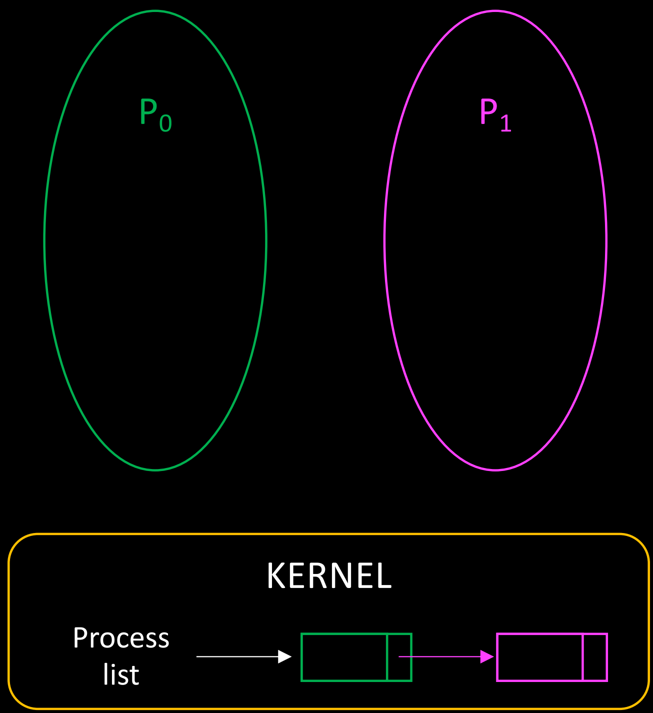
Process Creation



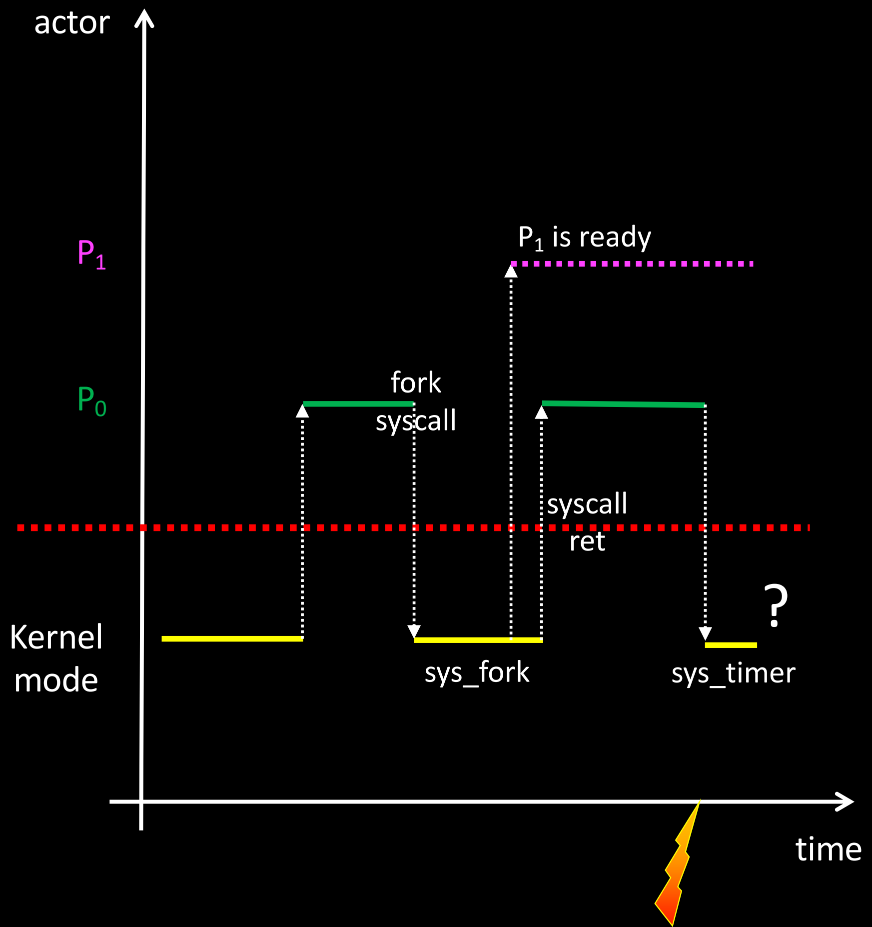


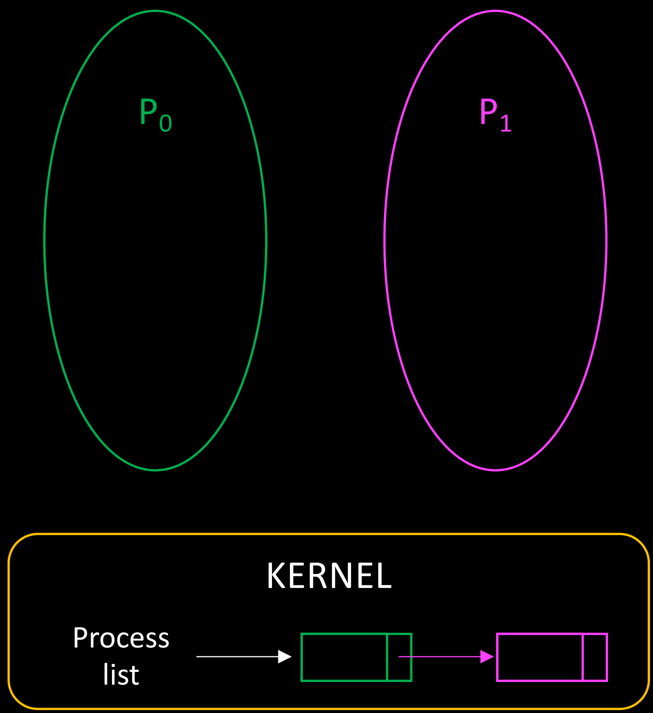
Process Creation



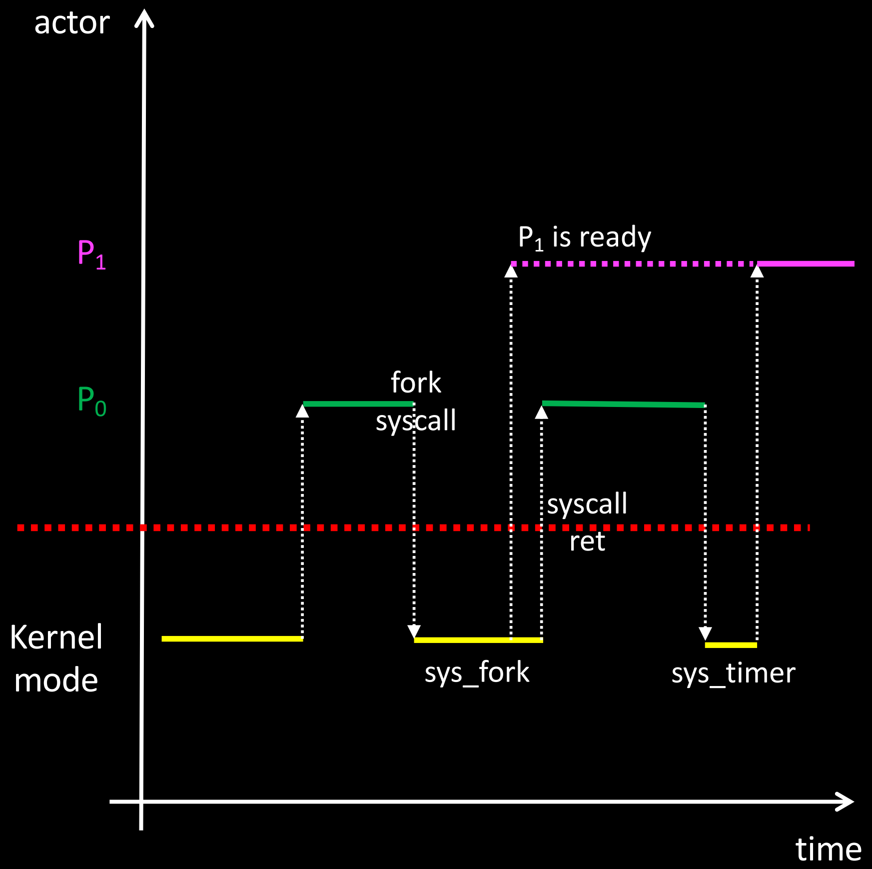


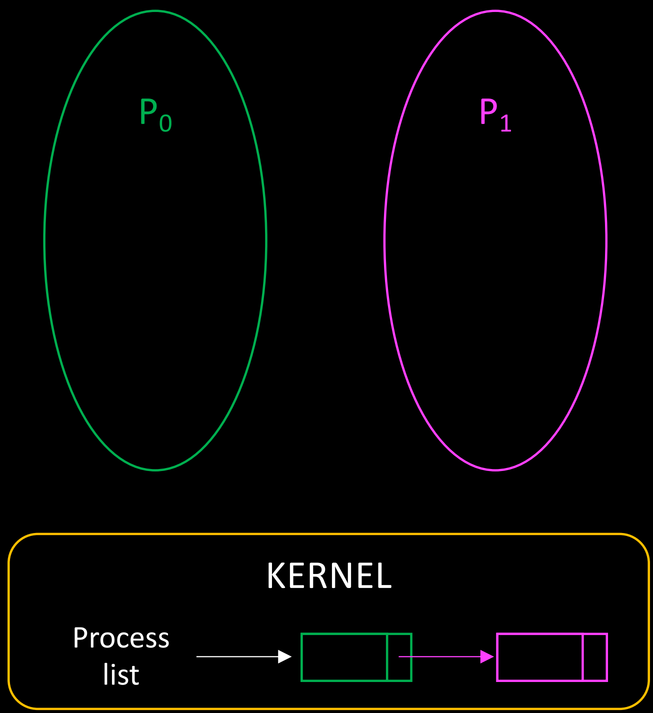
Process Creation





Process Creation





Process transformation

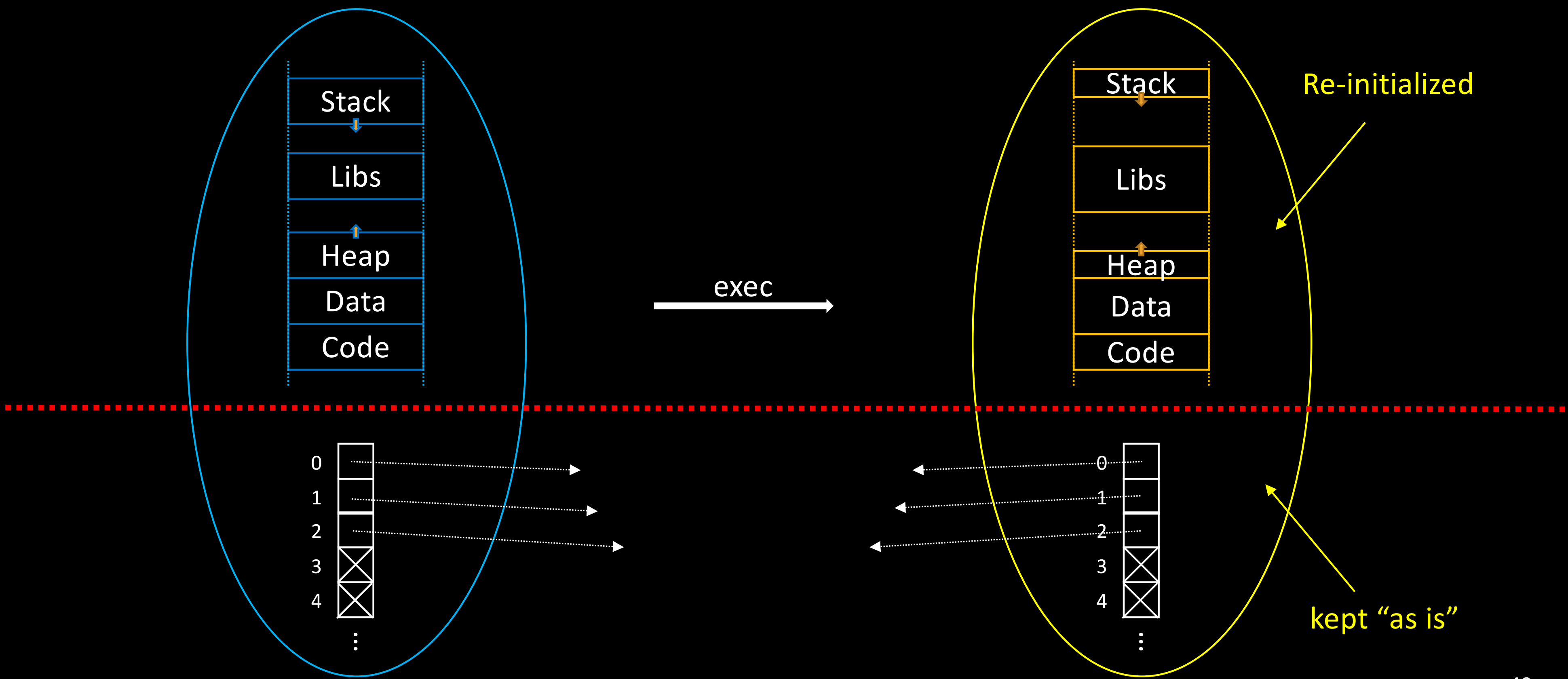
- A process can “reboot” and execute a new program
- Family of “`exec`” functions
  - `int execlp(char *file, char *arg0, ... , NULL);`
    - l: list of arguments
    - p: path
  - `int execvp(const char *file, char *const argv[]);`
    - v: vector of arguments
    - p: path
  - Use `execl` when list of arguments is known at compile time
    - Otherwise use `execv`

Process transformation

- Exec is a one-way trip
  - No return

```
int main (int argc, char *argv[])
{
  printf ("I am about to become ls -l\n");

  execl ("/bin/ls", "ls", "-l", NULL);
  perror ("execl");

  return EXIT_FAILURE;
}
```

Exec only preserves kernel information

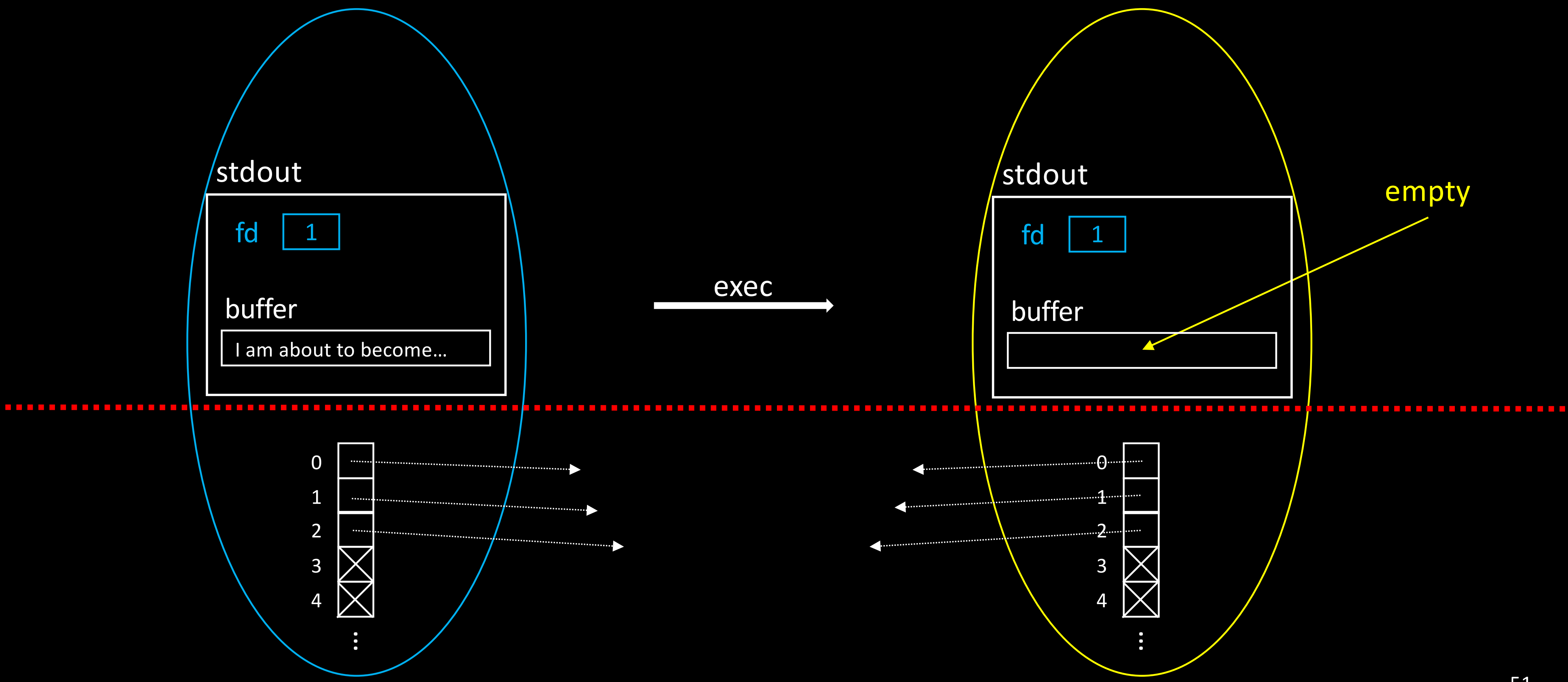


Process transformation

- Exec is a one-way trip
  - No return
- Caveat:
  - No visible printf ☹

```
int main (int argc, char *argv[])
{
  printf ("I am about to become ls -l");

  execl ("/bin/ls", "ls", "-l", NULL);
  perror ("execl");

  return EXIT_FAILURE;
}
```

Exec only preserves kernel information



Process transformation

- The file descriptor table is kept unmodified by exec
  - Redirections performed before exec are still in place
  - That's how we can redirect input/output of a binary program
    - No modification to the code of `ls`

```
char *FILENAME="output.txt";

int main (int argc, char *argv[])
{
  int fd = open (FILENAME, O_WRONLY | O_CREAT,
                 0666);
  check (fd, "Cannot open %s file", FILENAME);

  dup2 (fd, 1); close (fd);

  printf ("I am about to become "
          "\"ls -l > output.txt\"\n");

  execl ("/bin/ls", "ls", "-l", NULL);
  perror ("execl");

  return EXIT_FAILURE;
}
```

Process transformation

- The file descriptor table is kept unmodified by exec
  - Redirections performed before exec are still in place
  - That's how we can redirect input/output of a binary program
    - No modification to the code of `ls`
  - Oh, by the way
    - Do we see the output of `printf` this time?

```
char *FILENAME="output.txt";

int main (int argc, char *argv[])
{
  int fd = open (FILENAME, O_WRONLY | O_CREAT,
                 0666);
  check (fd, "Cannot open %s file", FILENAME);

  dup2 (fd, 1); close (fd);

  printf ("I am about to become "
          "\"ls -l > output.txt\"\n");

  execl ("/bin/ls", "ls", "-l", NULL);
  perror ("execl");

  return EXIT_FAILURE;
}
```

Combining fork() and exec()

- When the shell executes

  `ls -l > output.txt`

  - It cannot just
    - Redirect STDOUT to "output.txt"
    - And perform exec "ls"…
    - Because the shell wouldn't survive

- That's why the shell forks a child which will do the job

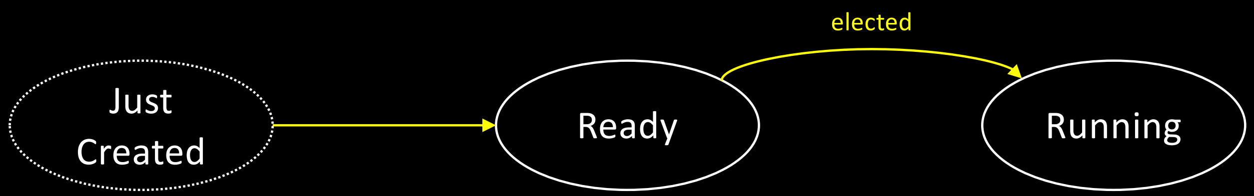
```
int main (int argc, char *argv[])
{
  pid_t pid;

  pid = fork ();
  if (pid) { // father

    // wait child

  } else { // Child

    // set redirections
    // exec command
  }

  return 0;
}
```

Process States

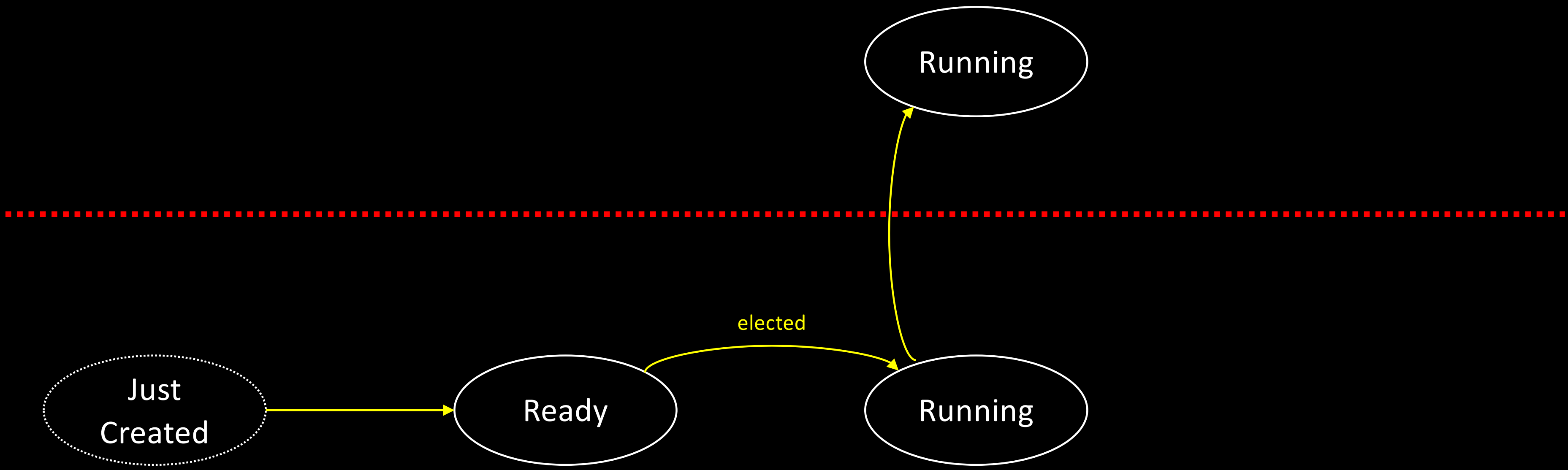
Just
Created

Process States

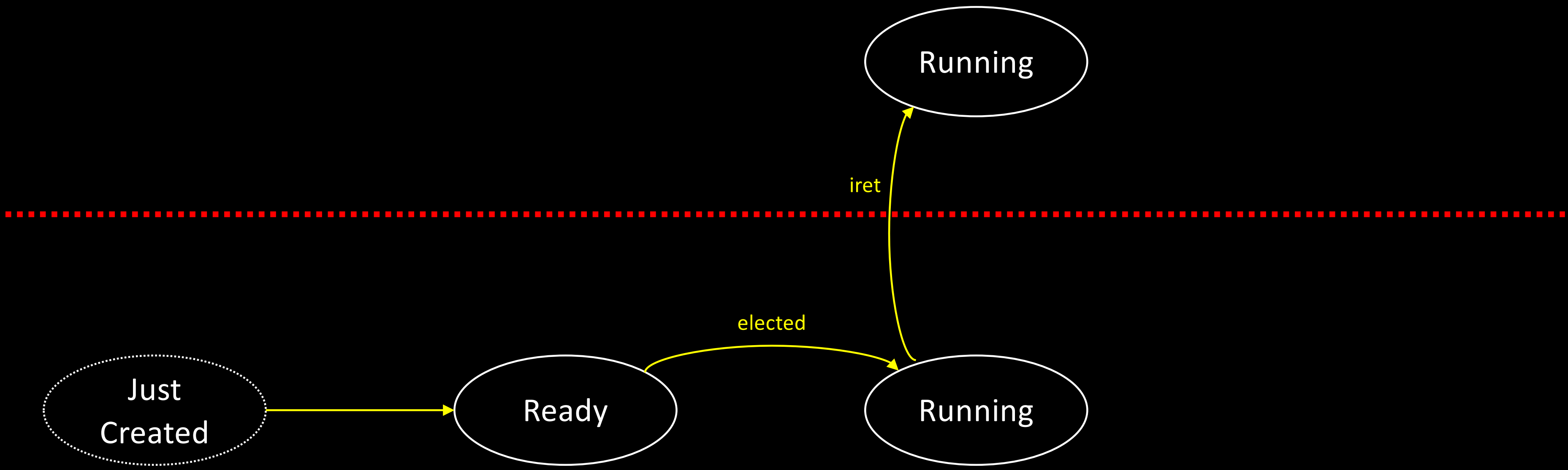
Process States



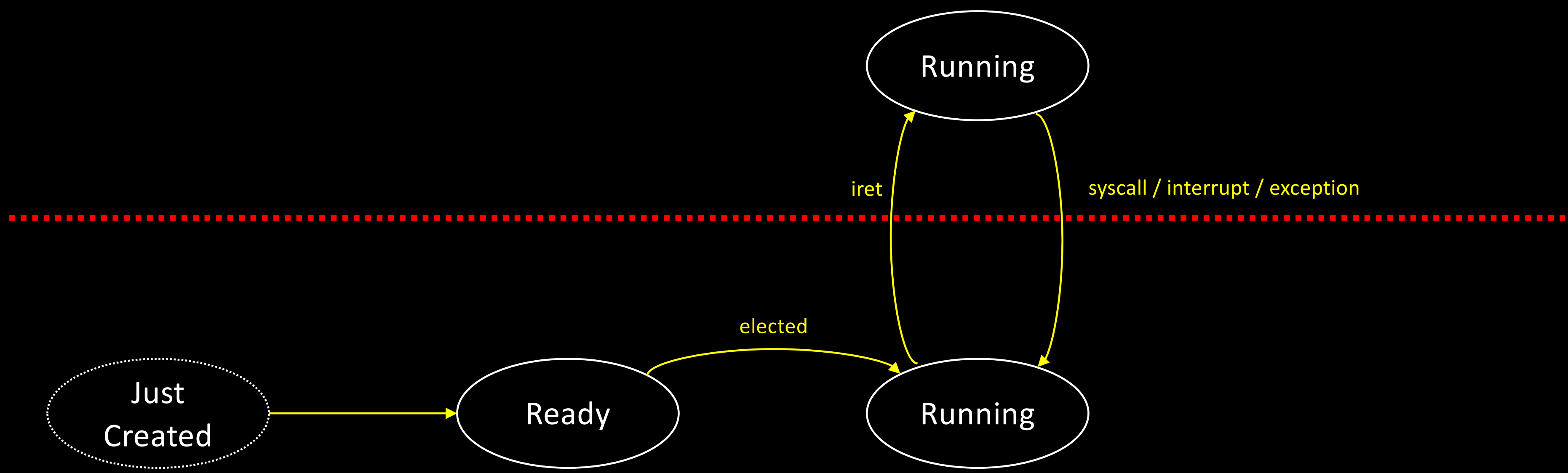
Process States



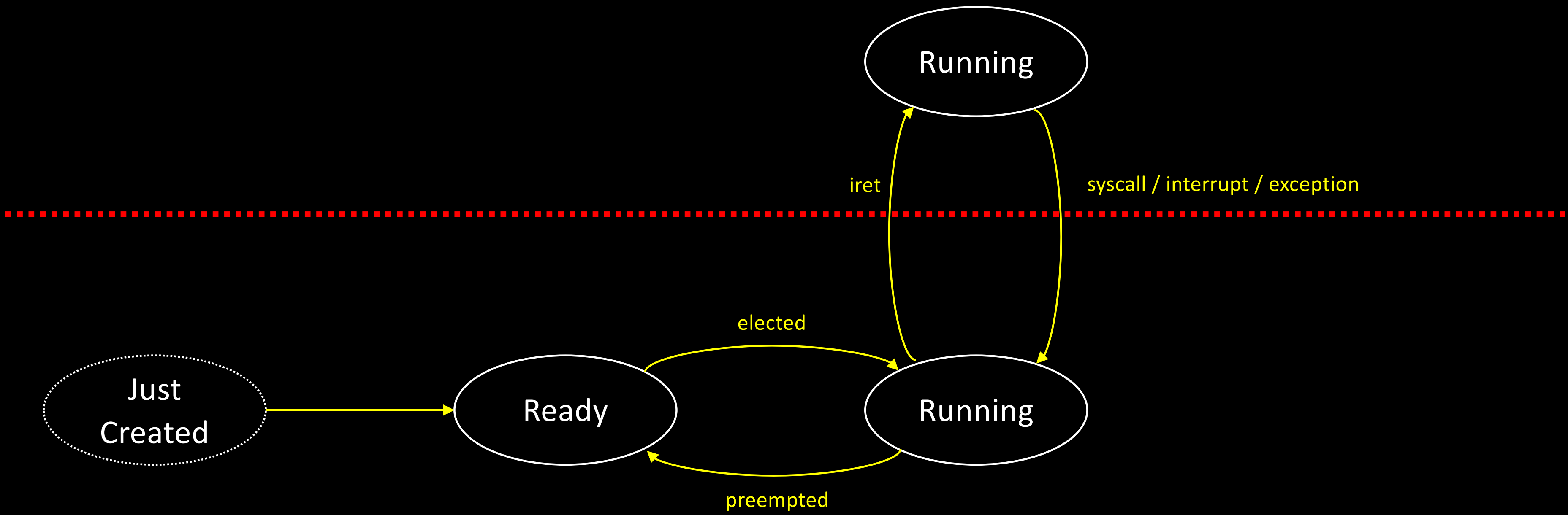
Process States



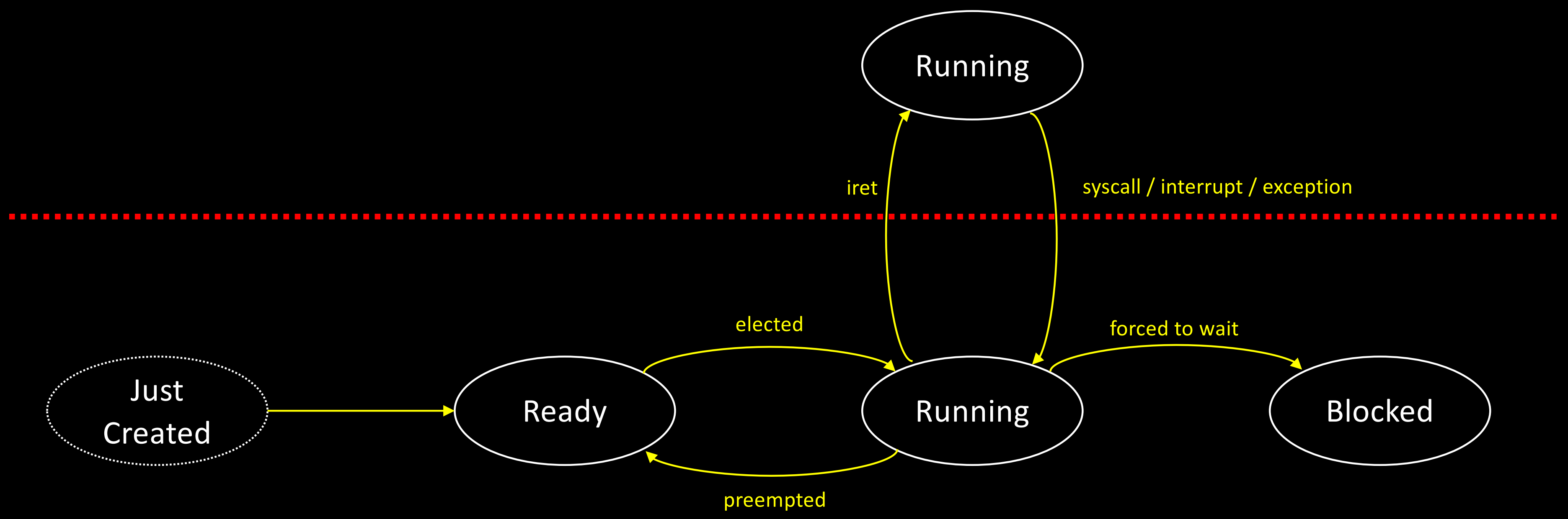
Process States



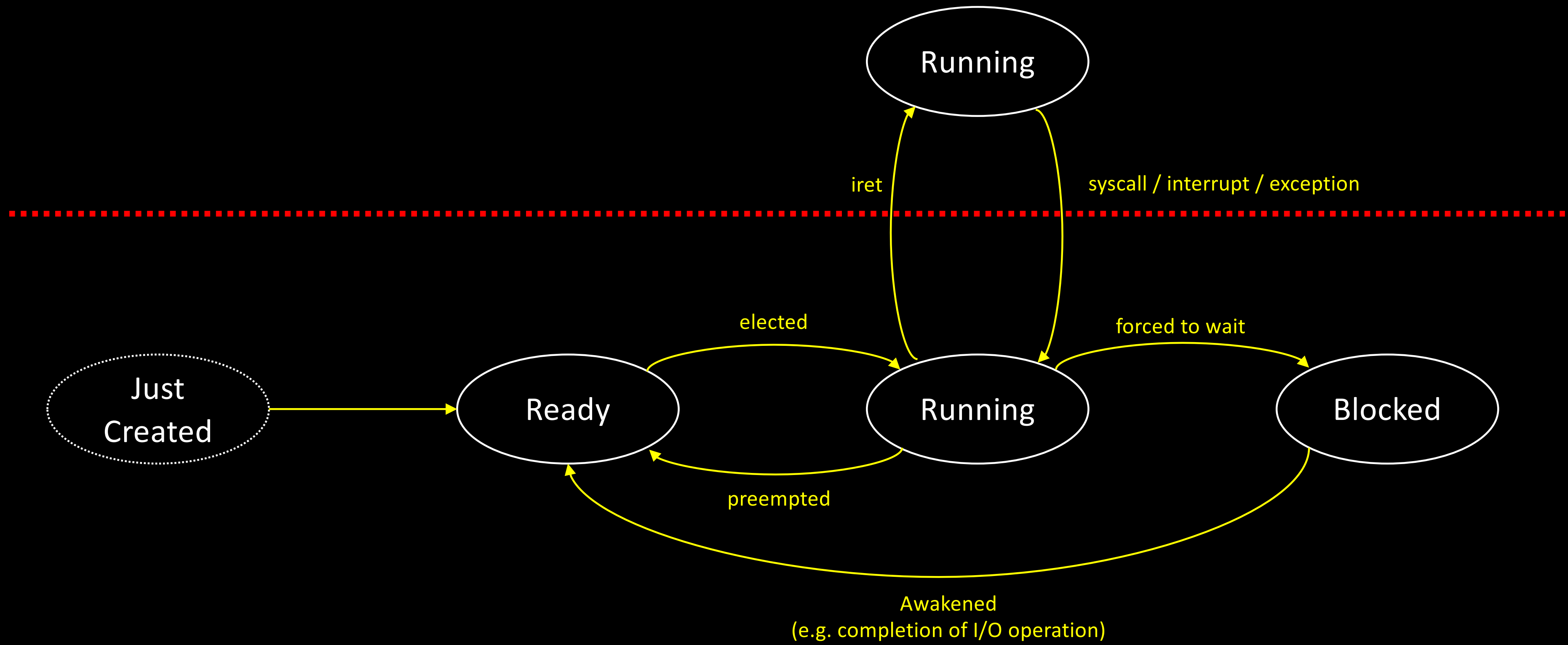
Process States



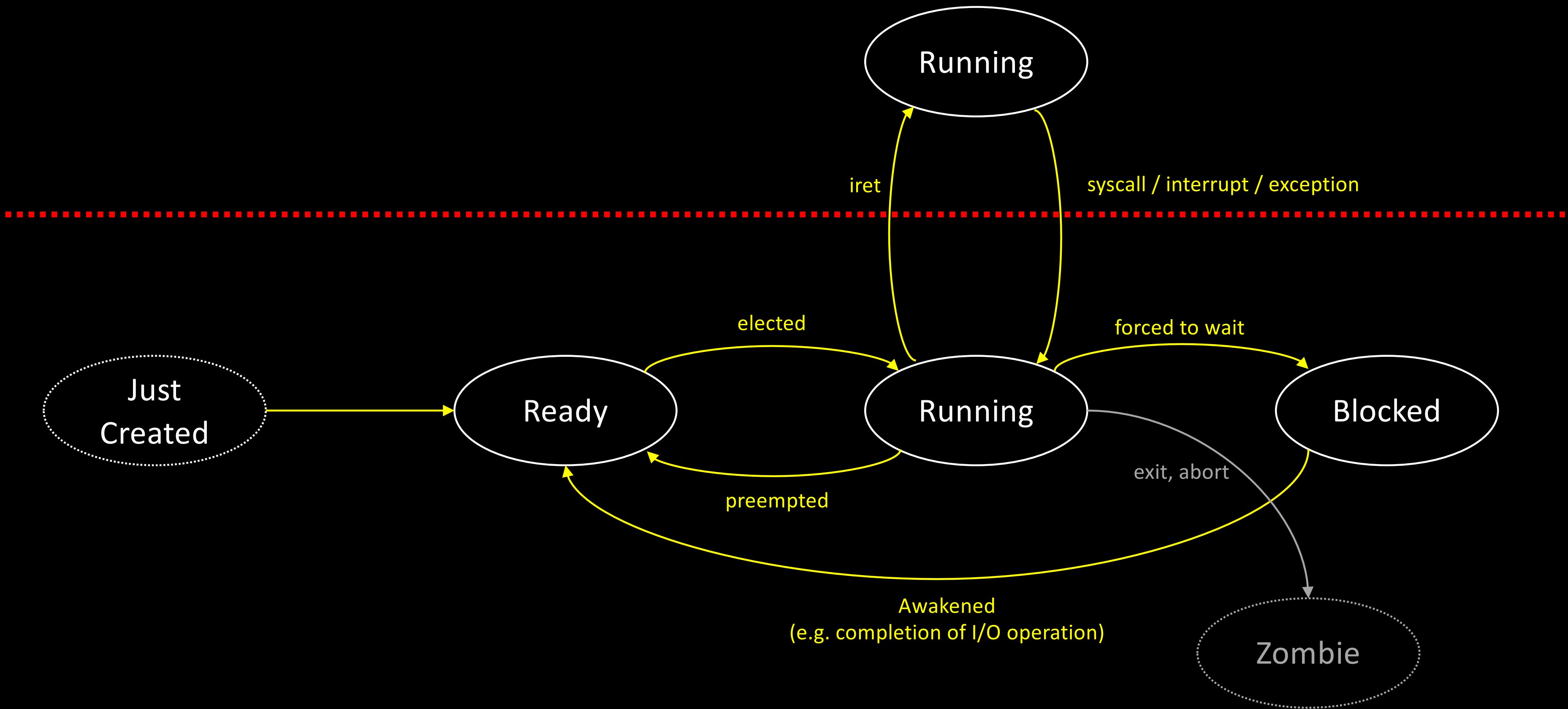
Process States



Process States



Process States



Additional resources available on http://gforgeron.gitlab.io/progsys/