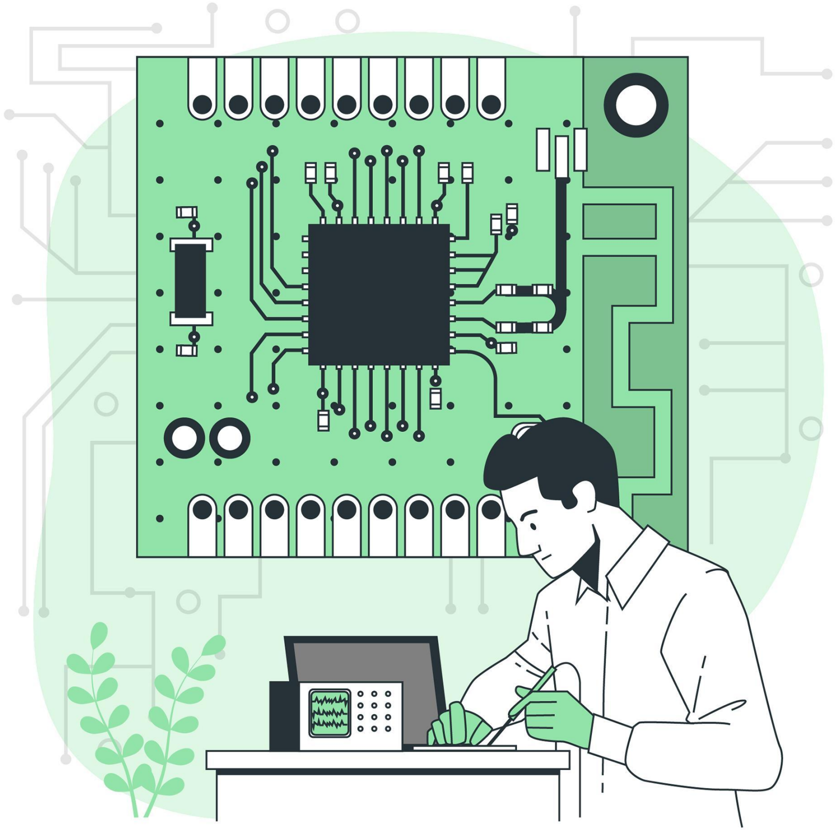
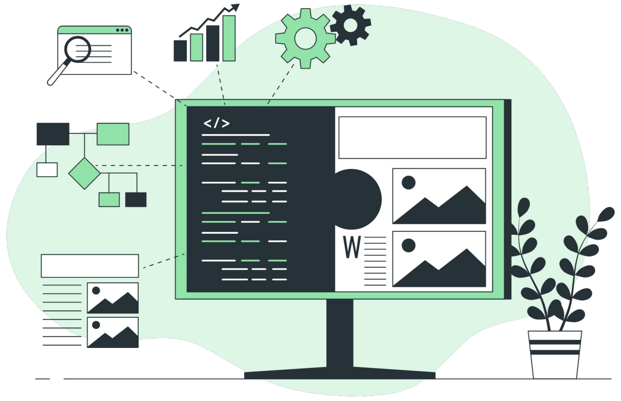
02 Süreçler

(Processes)



Process Nedir?

- Soyutlama
- Program == Process ?
- Çalışma zamanı
- Bir program diskte saklanabilir, process?



Neden Önemli?

İletişim

Kaynak İzolasyonu

Çoklu Görev

Process Oluşturma Süreci

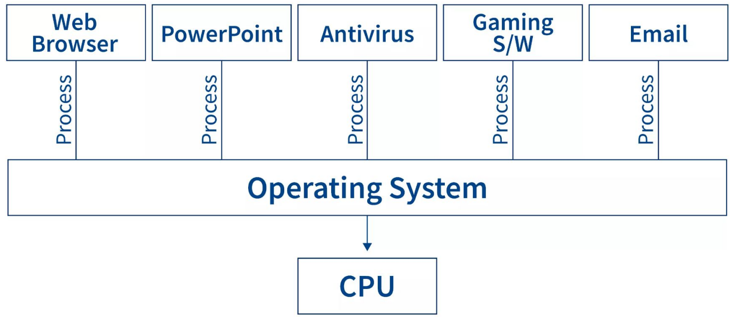
Fork

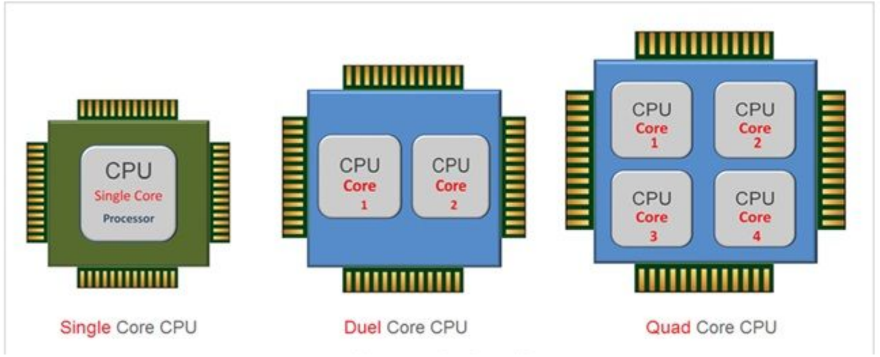
Execve

CreateProcess

Process ve Program Farkı

Çoklu Programlama ve Çekirdek



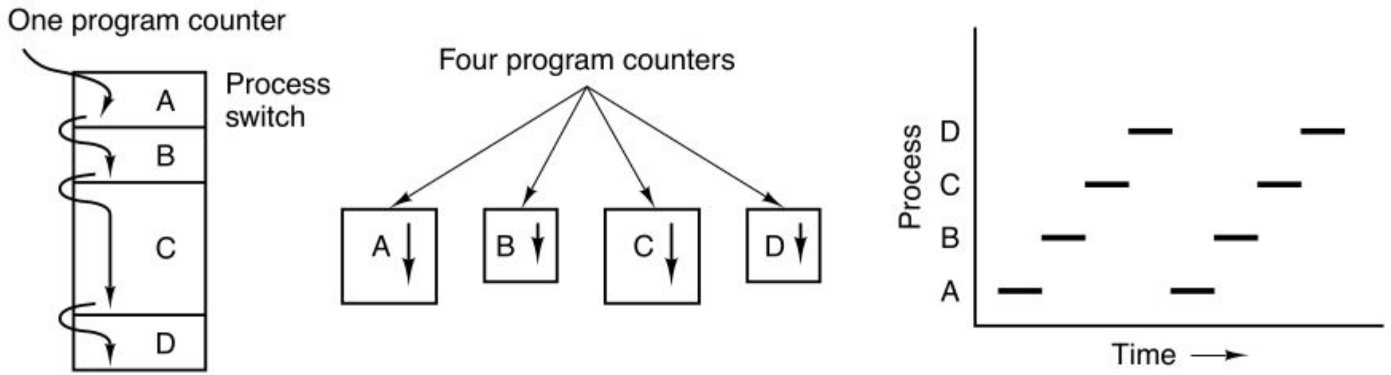


Çoklu Programlama ve Çekirdek

1 Çekirdek

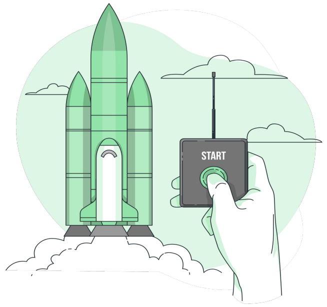
4 Çekirdek

Process Modeli

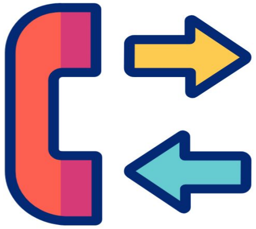


(a) (b) (c)

Process Oluşturma

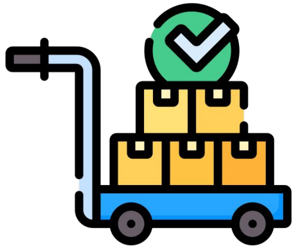


Sistem Başlangıcı



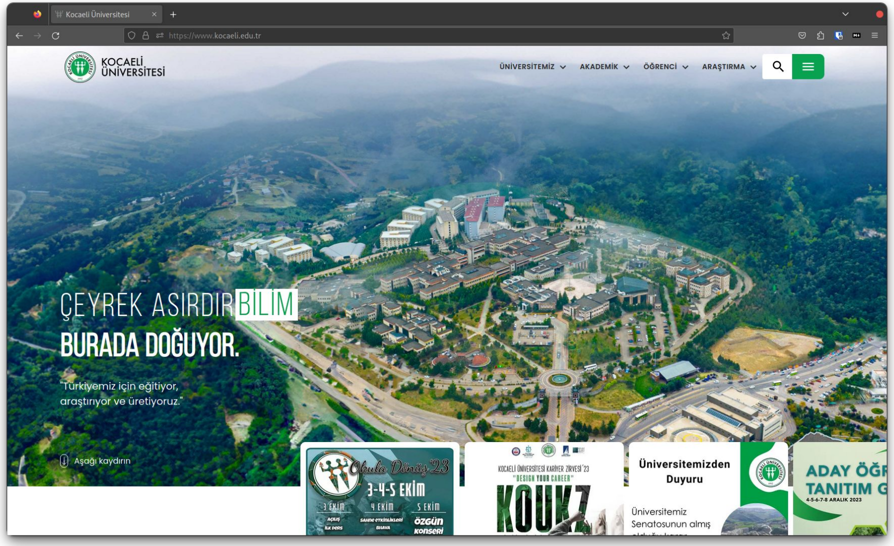
Sistem Çağrısı

Kullanıcı Talebi



Toplu Çağrı

Ön Plan ve Arka Plan Süreçleri



Process Sonlandırma

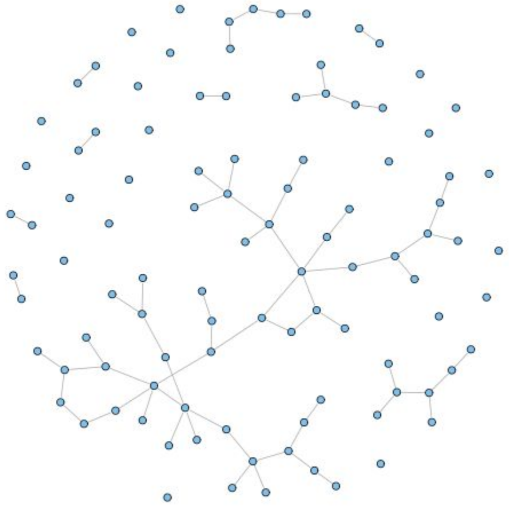
Normal Çıkış

Hata Nedeniyle

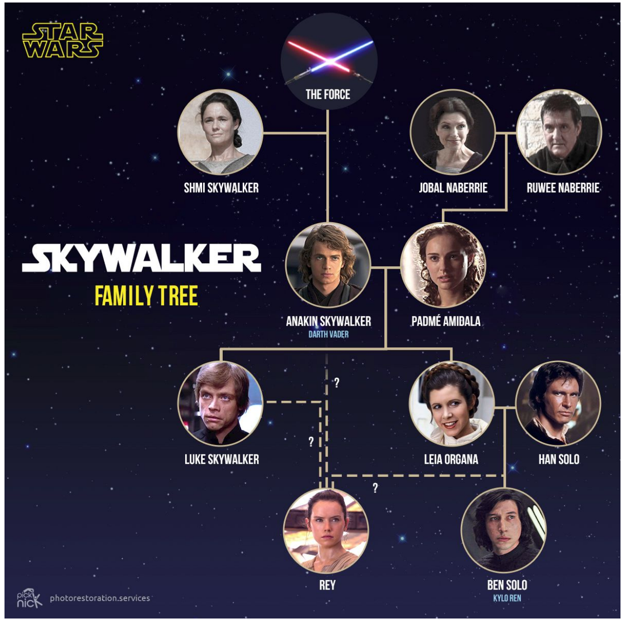
Ölümcül Hata

Başka Bir Süreç

Process Hiyerarşisi

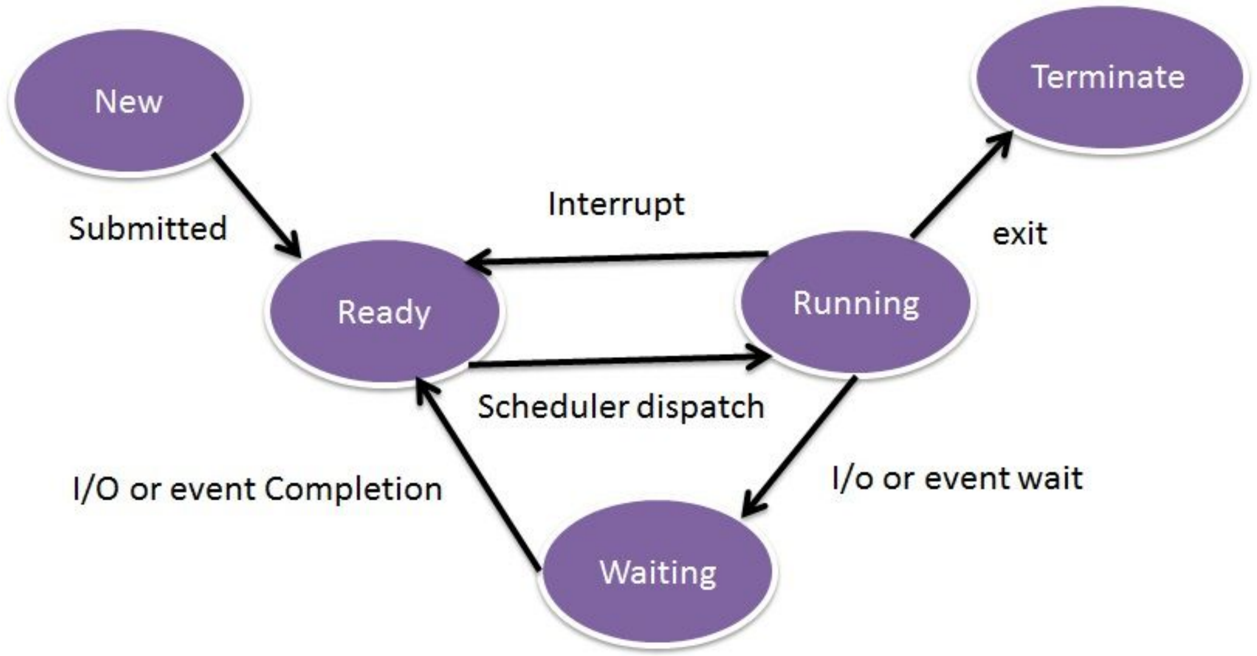


Windows

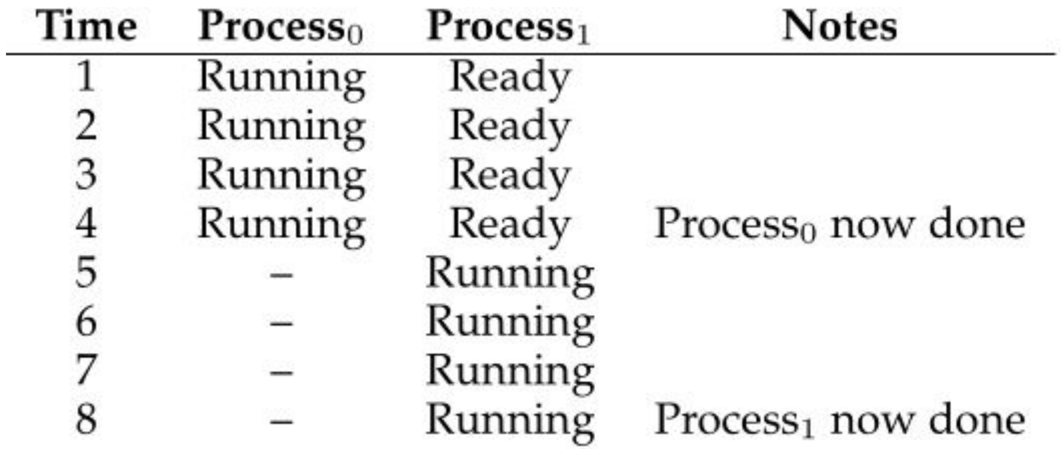


Unix

Process Durumları

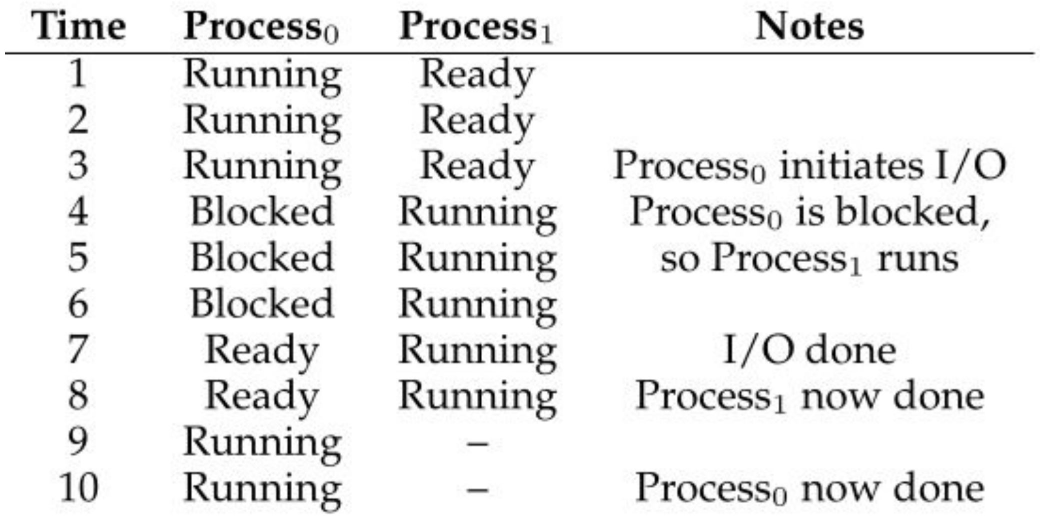


Process Durumları



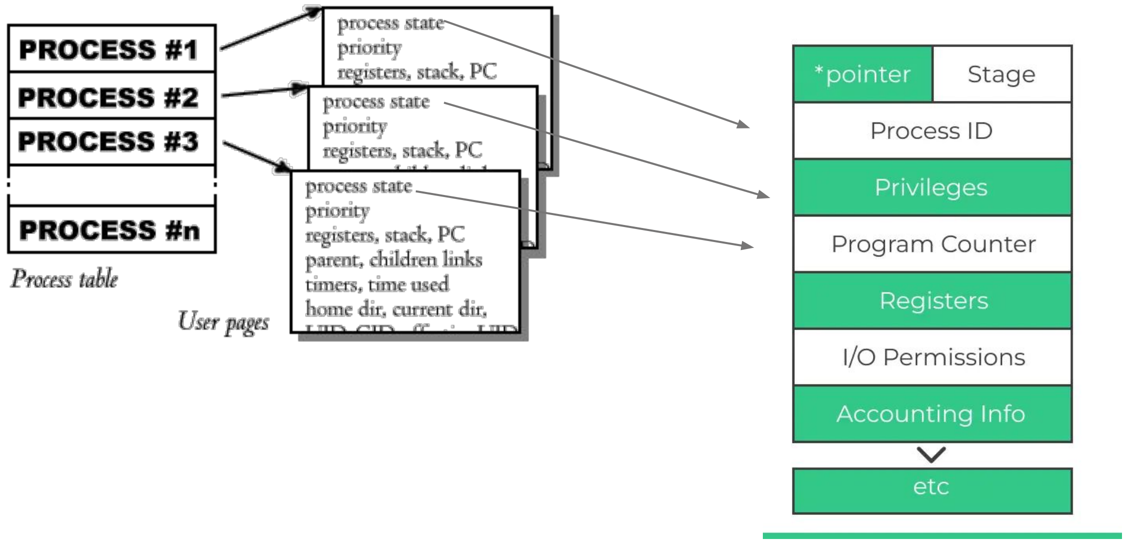
Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process_0 now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	Process_1 now done

Process Durumları

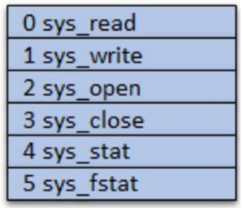


Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process_0 initiates I/O
4	Blocked	Running	Process_0 is blocked,
5	Blocked	Running	so Process_1 runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process_1 now done
9	Running	–	
10	Running	–	Process_0 now done

İşletim Sistemi Veri Yapıları



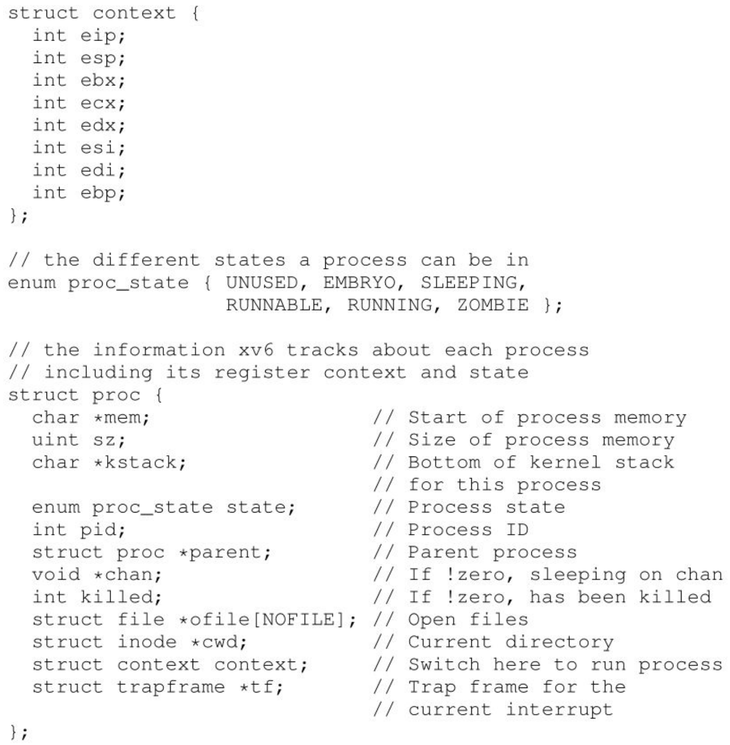
syscall table



0	sys_read
1	sys_write
2	sys_open
3	sys_close
4	sys_stat
5	sys_fstat

Örnek Bir Süreç (xv6)

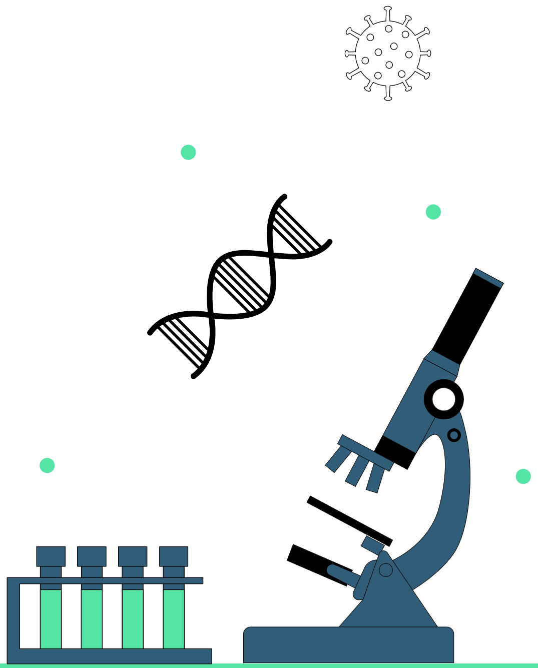
xv6 process header



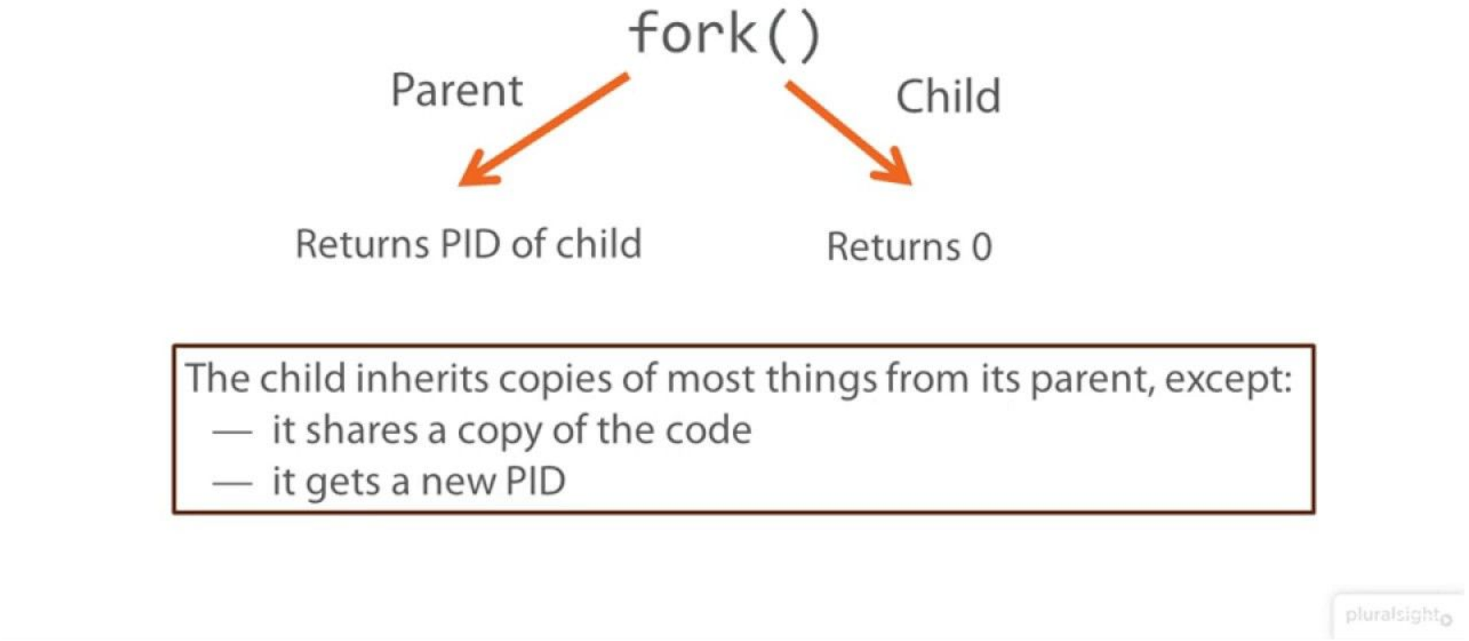
```
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If !zero, sleeping on chan
  int killed;                 // If !zero, has been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

02

Process API



Fork ()

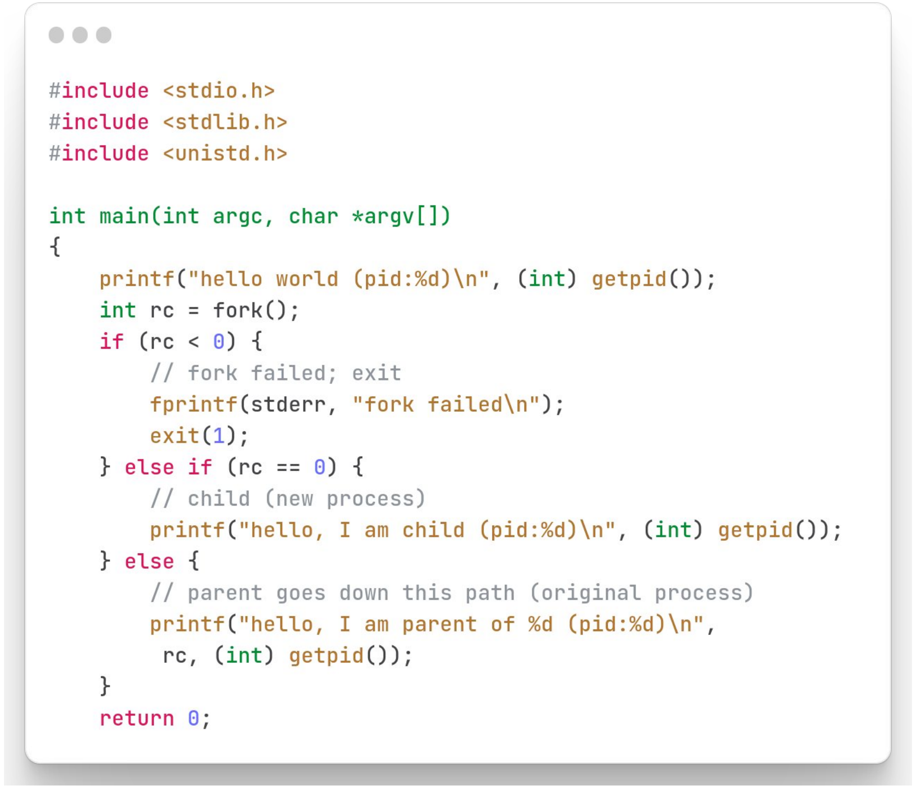


The child inherits copies of most things from its parent, except:

— it shares a copy of the code

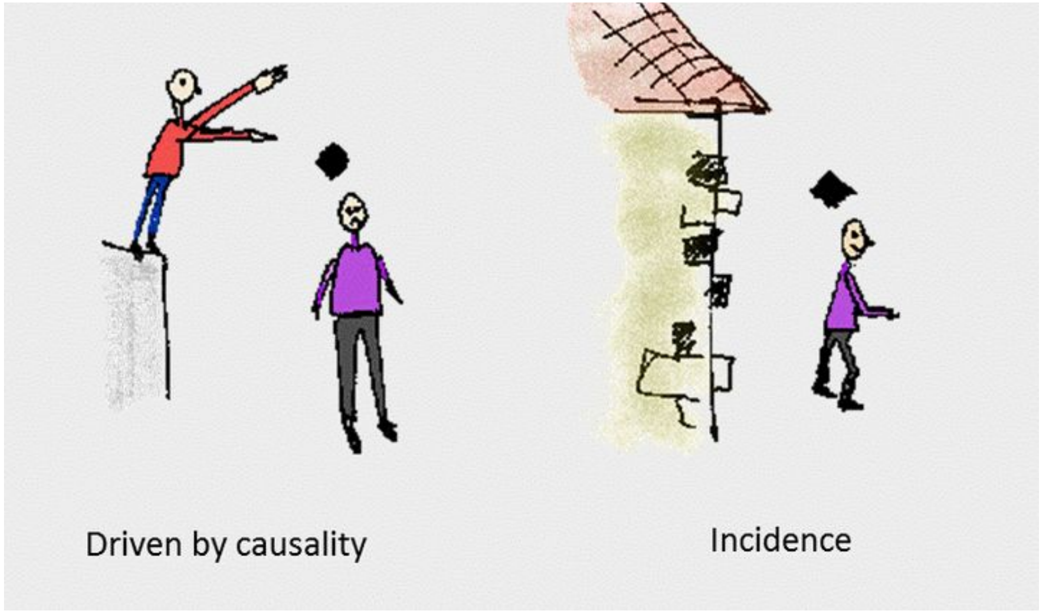
— it gets a new PID

Fork()

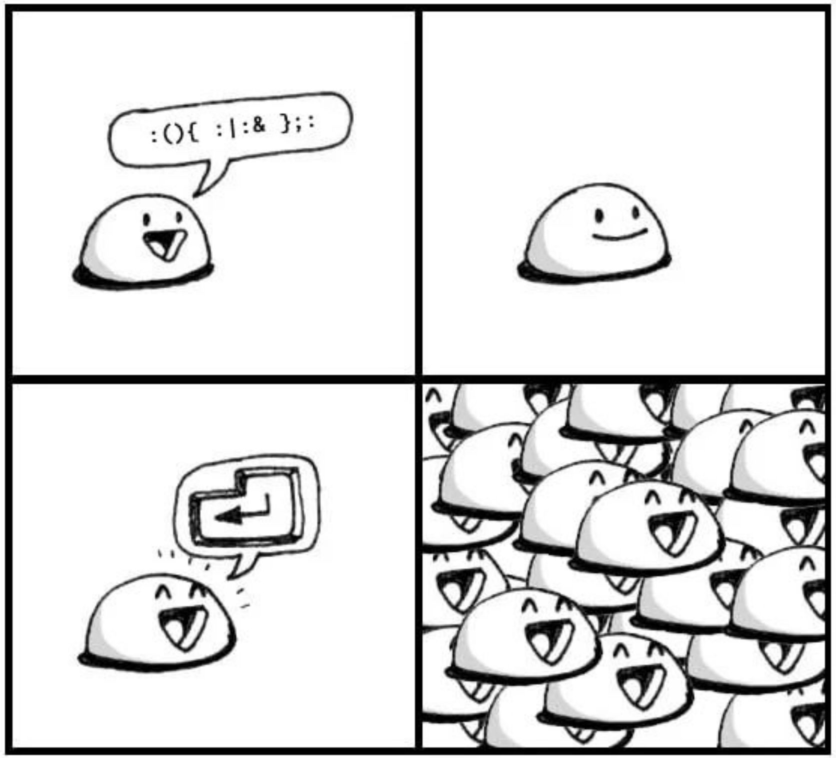


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
         rc, (int) getpid());
    }
    return 0;
```

Deterministik / Stokastik

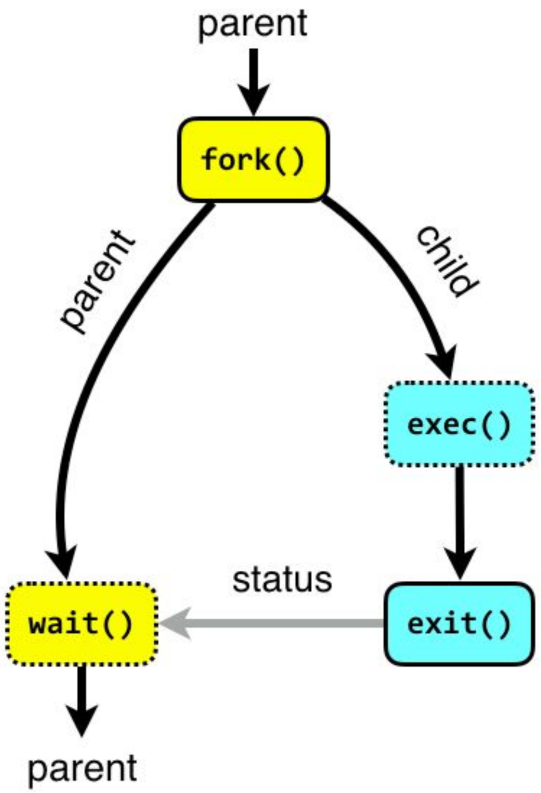


Fork Bomb 💣

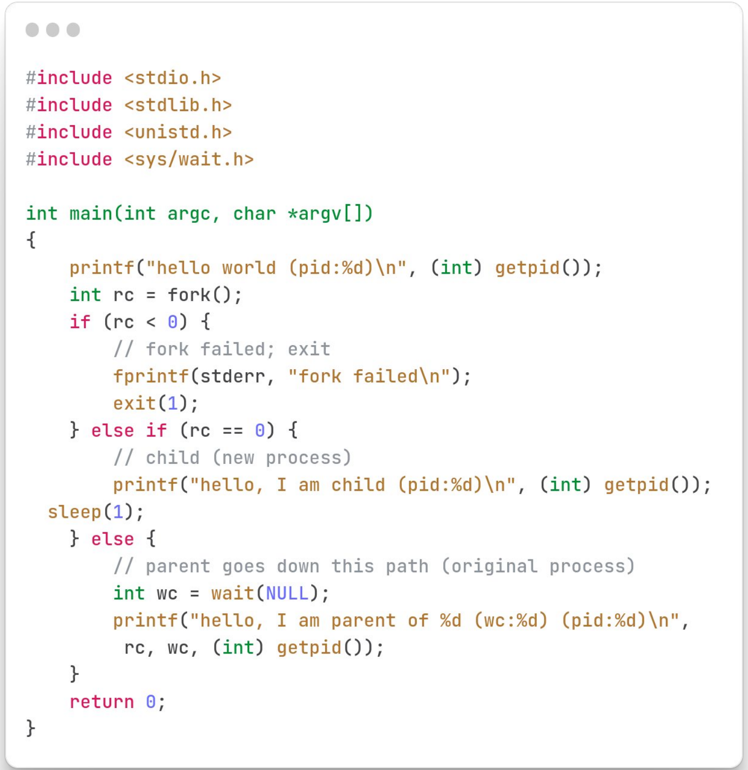


```
:(){ :|:& };:
```

wait()

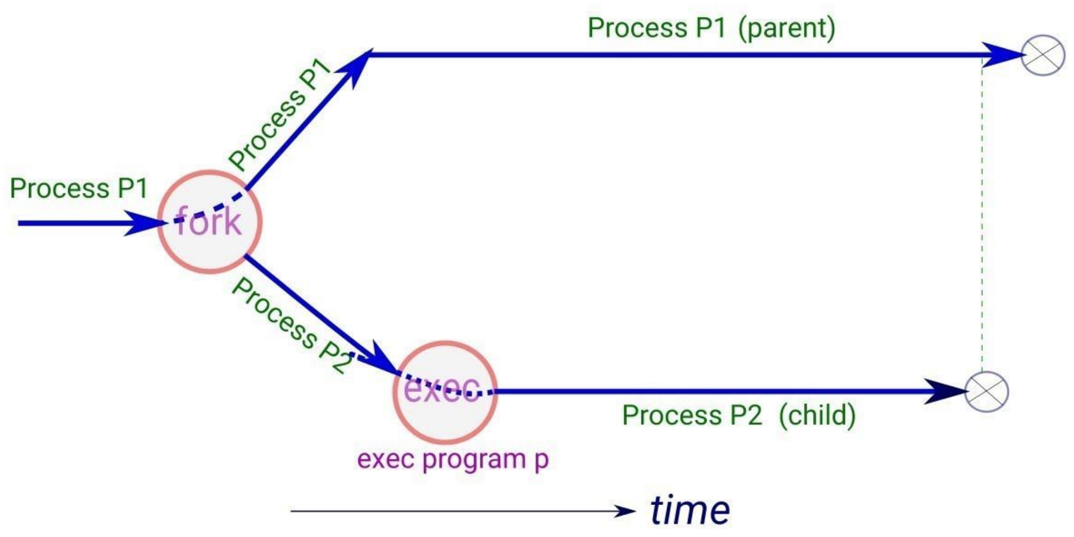


wait()

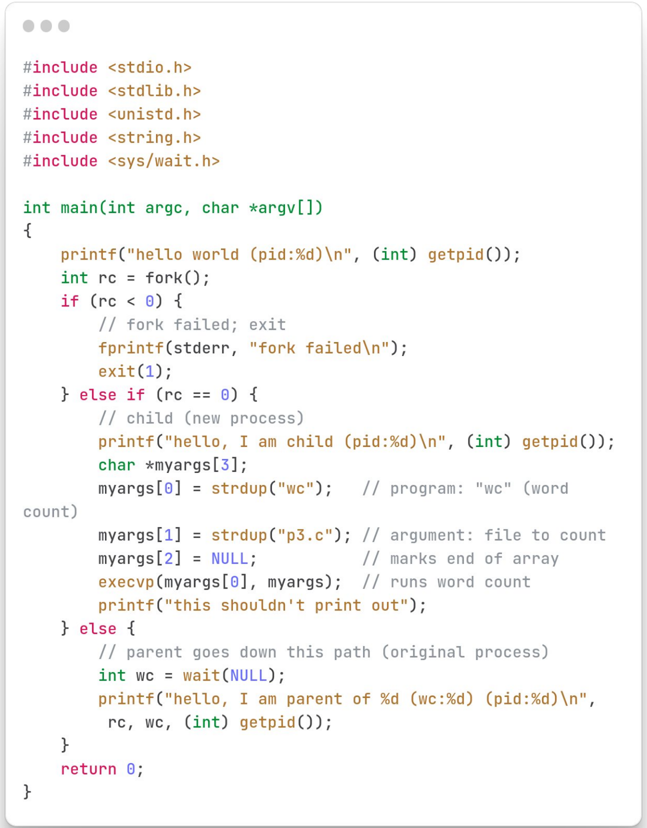


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
  sleep(1);
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
         rc, wc, (int) getpid());
    }
    return 0;
}
```

exec()


Process P1 (parent)
Process P1
Process P1
fork
Process P2
exec
Process P2 (child)
exec program p
time

exec()



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word
count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
         rc, wc, (int) getpid());
    }
    return 0;
}
```