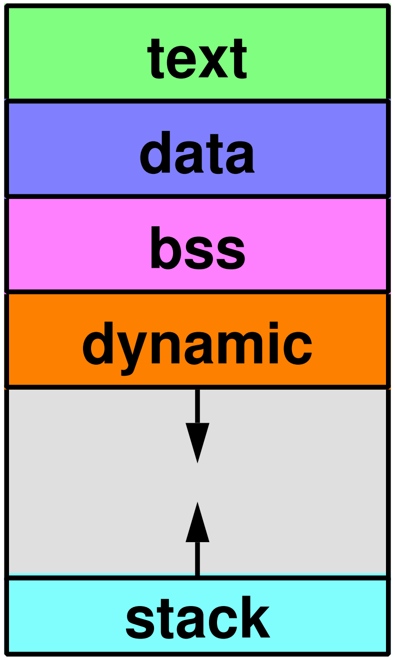
Modified Program

```
int nprimes;                    // in bss region
int *prime;                     // in bss region
int main(int argc, char *argv[]) { // in stack
  int i;                        // in stack
  int current = 2;              // in stack
  nprimes = atoi(argv[1]);
  prime = (int*)malloc(nprimes*sizeof(int));
  prime[0] = current;
  for (i=1; i<nprimes; i++) {
     ...
  }
  return(0);
}
```



- where do all the variables reside?
- what is `argv[1]` and why `atoi()`?
- what is `sizeof()`?
- what does `malloc()` do?

1.3 A Simple OS

- **OS Structure**
- **Processes, Address Spaces, & Threads**
- ***Managing Processes***
- **Loading Program Into Processes**
- **Files**

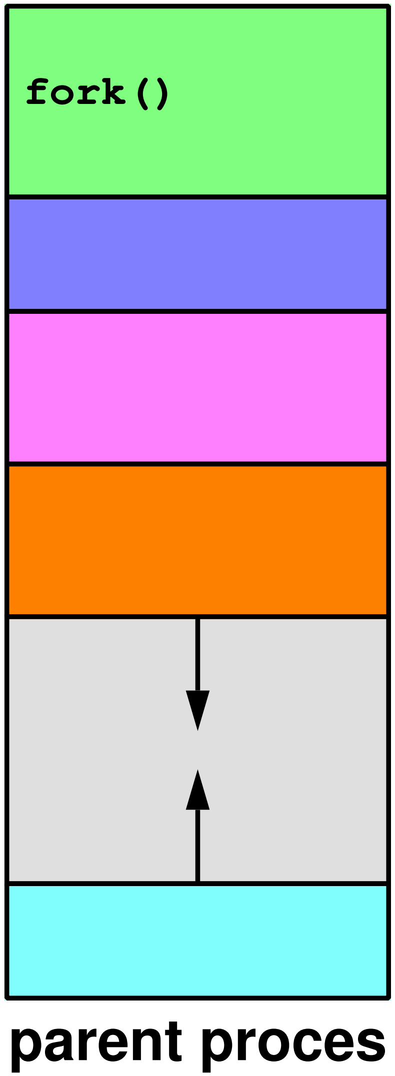
Program Execution

- With abstraction, comes an interface / API
  - for processes
    - `fork(), exit(), wait(), exec()`
      - it's very important to understand ***what they do exactly*** because you will implement them in kernel assignments

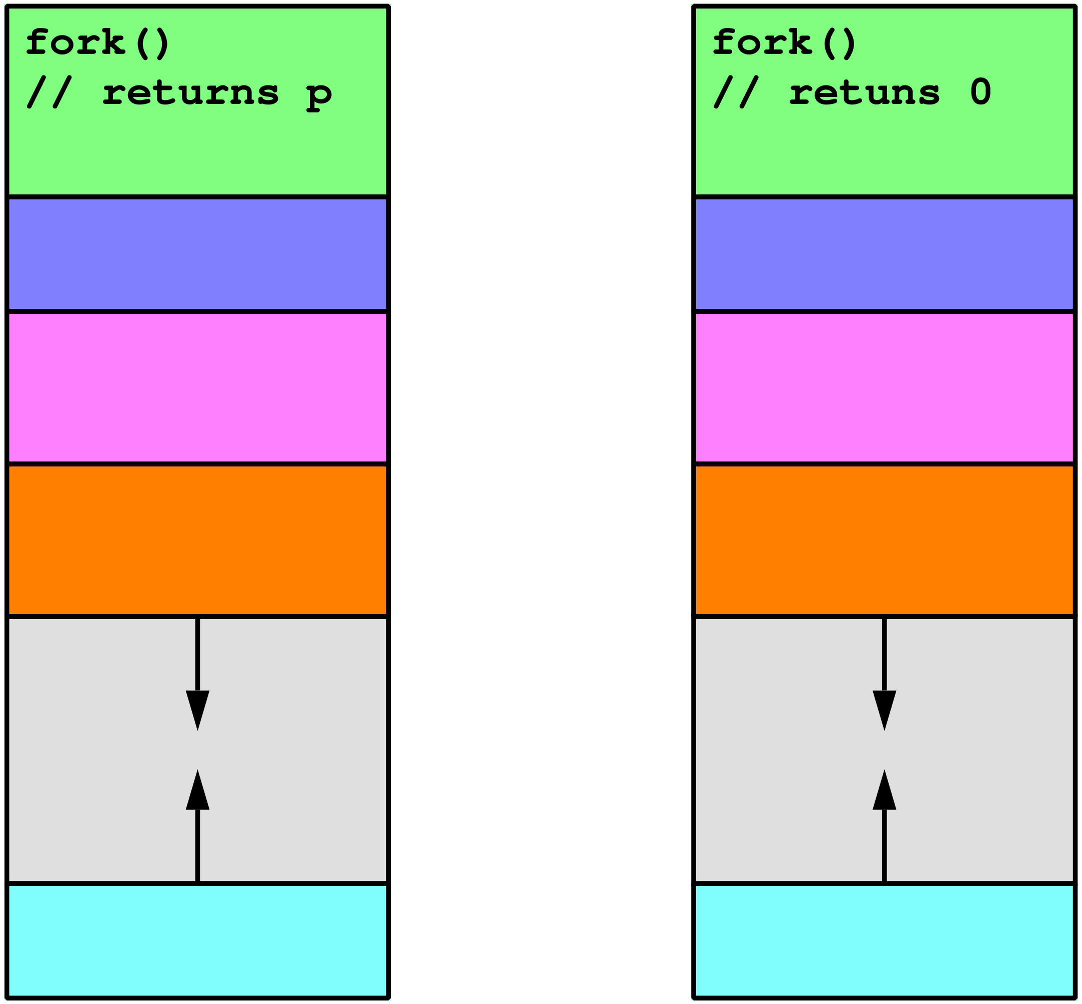
Creating a Process

- Creating a process is deceptively simple
  - make a copy of a process (the parent process)
    - `pid_t fork(void)`
    - the process where `fork()` is called is the *parent* process
    - the copy is the *child* process
    - in a way, `fork()` returns twice
      - once in the parent, the returned value is the *process ID (PID)* of the child process
      - once in the child, the returned value is 0
      - a PID is 16-bit long
  - this is the *only* way to create a process
- Making a copy of the entire address space can be expensive
  - Ch 7 shows speed up tricks
  - e.g., text segment is read-only so parent and child can share it
- Example: relationship between a shell (i.e., a command interpreter, such as `/bin/tcsh`) and `/bin/ls`

Creating a Process: Before



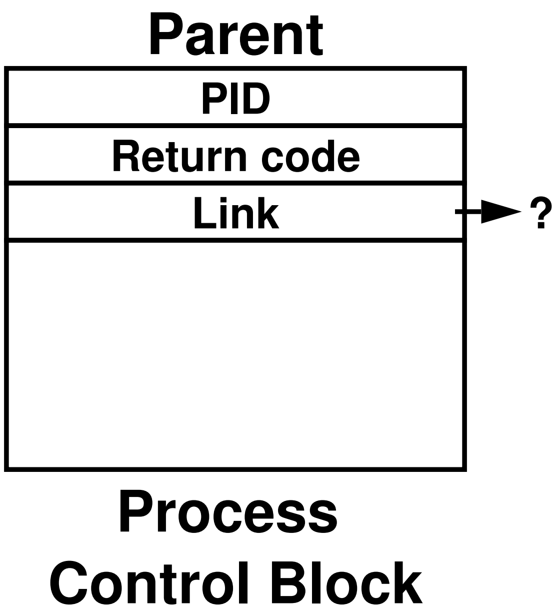
Creating a Process: After



parent proces

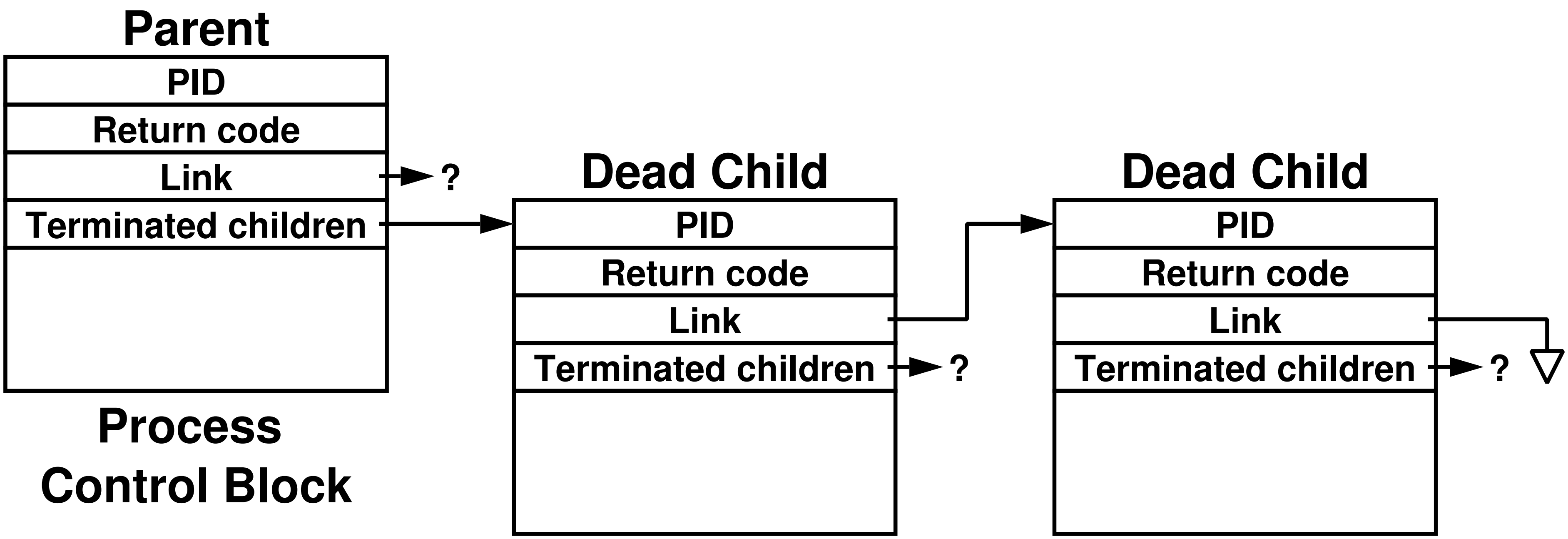
child proces
(pid = p)

Process Control Blocks



- ***Process Control Block (PCB)*** **is a kernel data structure**
  - **pretty much every field is unsigned**
  - **return code (when a process dies) is 8-bit long**
    - **so that the parent process can know what happened to child**
  - **the "Link" field points to the next PCB**
    - **but, the next PCB in what list?**

Process Control Blocks



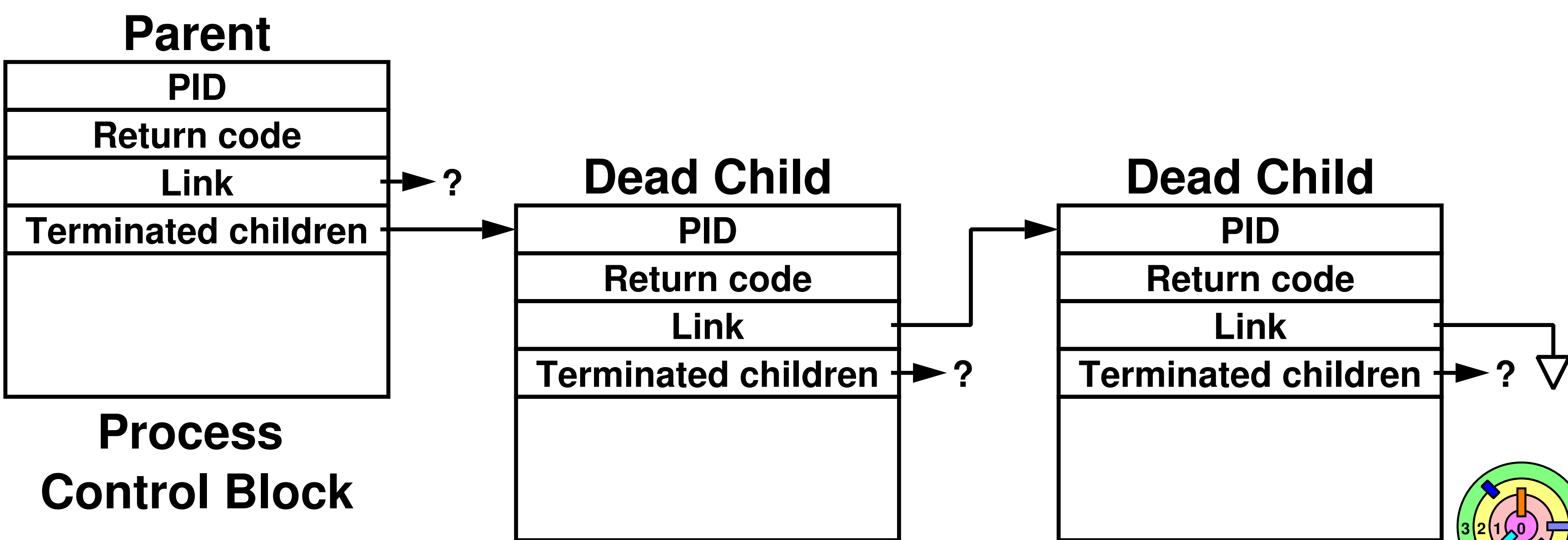
- ***Process Control Block (PCB)*** **is a kernel data structure**
  - **pretty much every field is unsigned**
  - **return code (when a process dies) is 8-bit long**
    - **so that the parent process can know what happened to child**
  - **the "Link" field points to the next PCB**
    - **but, the next PCB in what list?**
- **Above is *not* a real implementation (just an example)**

The `exit()` System Calls

- The `exit()` system call

```
void exit(int status)
```

  - your process can call `exit(n)` to self-terminate
    - set `n` to be the "exit/return code" of this process
    - this sytem call does not return (your process will die inside the kernel)



Process Control Block

The `exit()` System Calls

- The `exit()` system call

```
void exit(int status)
```

  - your process can call `exit(n)` to self-terminate
    - set `n` to be the "exit/return code" of this process
    - this sytem call does not return (your process will die inside the kernel)

- Where does the "primes" program go after it executes the "`return(0)`"?
  - it returns to a "startup" function
  - the code of the "startup" function is simply:

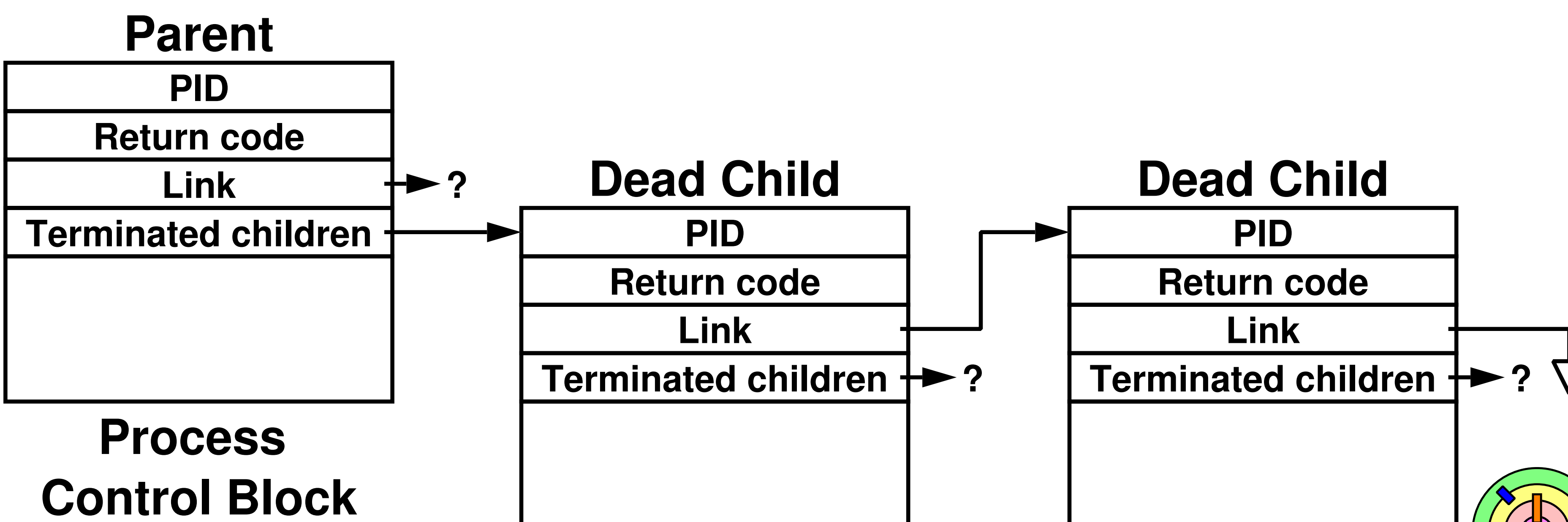
```
exit(main());
```

The `wait()` System Calls

- The `wait()` system call

```
pid_t wait(int *status)
```

  - your process can call `wait()` to wait for *any* child process to die
    - returns the PID of a dead child process where `(*status)` is the exit/return code of the corresponding child process
      - if there are more than one dead child processes, one of them will be *chosen at random*

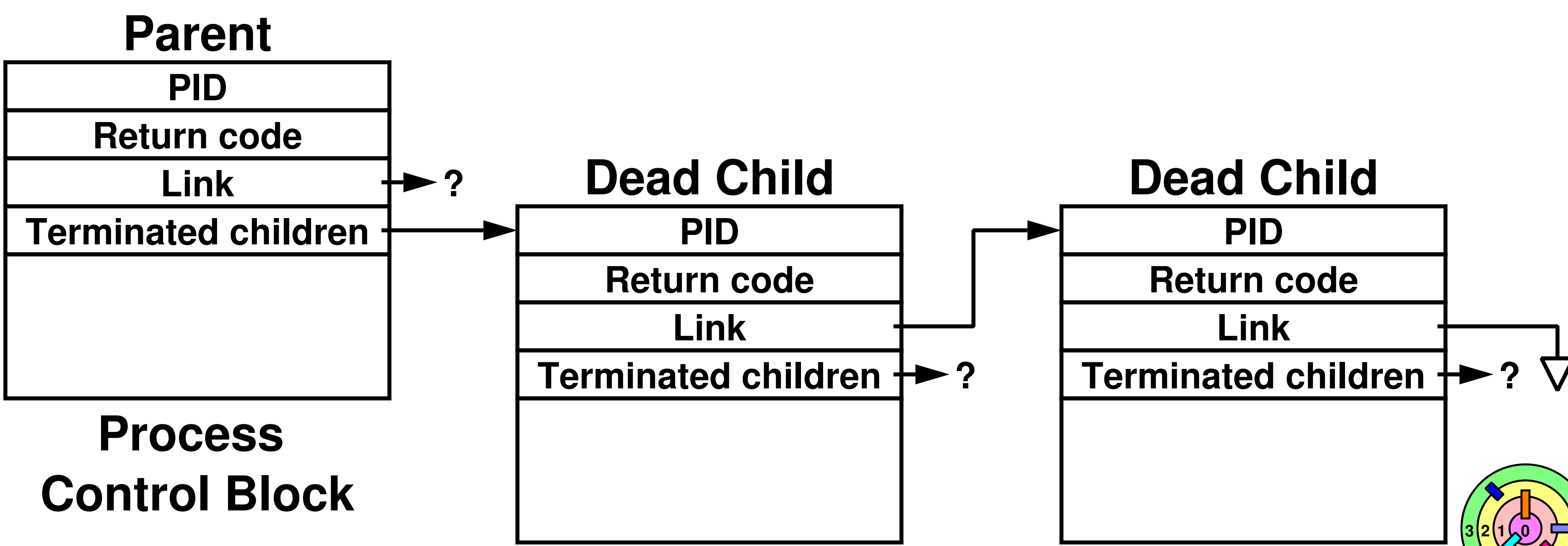


The `wait()` System Calls

- The `wait()` system call

```
pid_t wait(int *status)
```

  - your process can call `wait()` to wait for *any* child process to die
    - it's a *blocking call*, i.e., the calling process gets suspended inside the kernel if this call cannot return yet



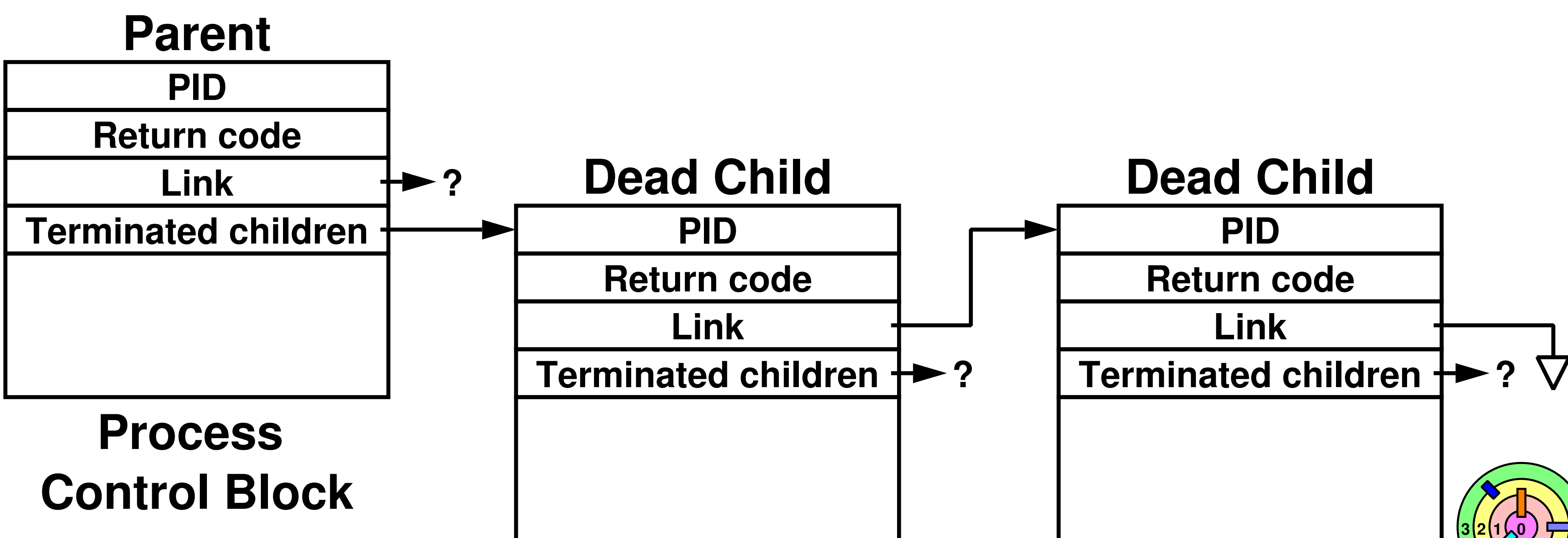
Fork and Wait

```
short pid;
if ((pid = fork()) == 0) {
  /* some code is here for the child to execute */
  exit(n);
} else {
  int ReturnCode;
  while(pid != wait(&ReturnCode))
     ;
  /* the child has terminated with ReturnCode as
     its return code */
}
```

- e.g., this is the first step when `/bin/tcsh` forks `/bin/ls`
- what does `exit(n)` do other than copying `n` into PCB?
  - least significant 8-bits of `n`
- what happens when `main()` calls `return(n)`?
  - eventually, `exit(n)` will be invoked
- `pid_t wait(int *status)` is a ***blocking call***
  - it reaps dead child processes ***one at a time***
- parent and child are the same "program" here!

Fork and Wait

```
short pid;
if ((pid = fork()) == 0) {
   /* some code is here for the child to execute */
   exit(n);
} else {
   int ReturnCode;
   while(pid != wait(&ReturnCode))
       ;
   /* the child has terminated with ReturnCode as
      its return code */
}
```



Fork and Wait

```
short pid;
if ((pid = fork()) == 0) {
  /* some code is here for the child to execute */
  exit(n);
} else {
  int ReturnCode;
  while(pid != wait(&ReturnCode))
     ;
  /* the child has terminated with ReturnCode as
     its return code */
}
```

- What if you don't want to write your code this way?
  - you can write any code you want, you just shouldn't expect your code to work if you write weird code
  - you need to understand exactly what these system calls do and use them appropriately
    - if you do something weird, the OS will try to satisfy your request, but may end up with results you don't expect

Process Termination Issues

- PID is only 16-bits long
  - OS must not reuse PID too quickly or there may be ambiguity
- When `exit()` is called, the OS must not free up PCB too quickly
  - parent needs to get the return code
  - it's okay to free up everything else (such as address space)
- Solutions for both is for the terminated child process to go into a *zombie* state
  - only after `wait()` returned with the child's PID can the PID be reused and the PCB can be freed up
  - but what if the parent calls `exit()` while the child is in the zombie state?
    - process 1 (the process with PID=1) inherits all the children of this parent process
      - this is known as *"reparenting"*
    - process 1 keeps calling `wait()` to reap the zombies

1.3 A Simple OS

- OS Structure
- Processes, Address Spaces, & Threads
- Managing Processes
- *Loading Program Into Processes*
- Files

Loading Programs Into Processes

- How do you run a program?
  - make a copy of a process
    - any process
  - replace the child process with a new one
    - wipe out the child process
      - not everything, some stuff survives this (i.e., won't get destroyed)
      - definitely need a *new address space* since we will be running a different program
    - using a family of system calls known as *exec*
  - kind of a waste to make a copy in the first place
    - but it's the only way
    - also, the OS does not know if the reason the parent process calls `fork()` is to run a new program or not

Exec

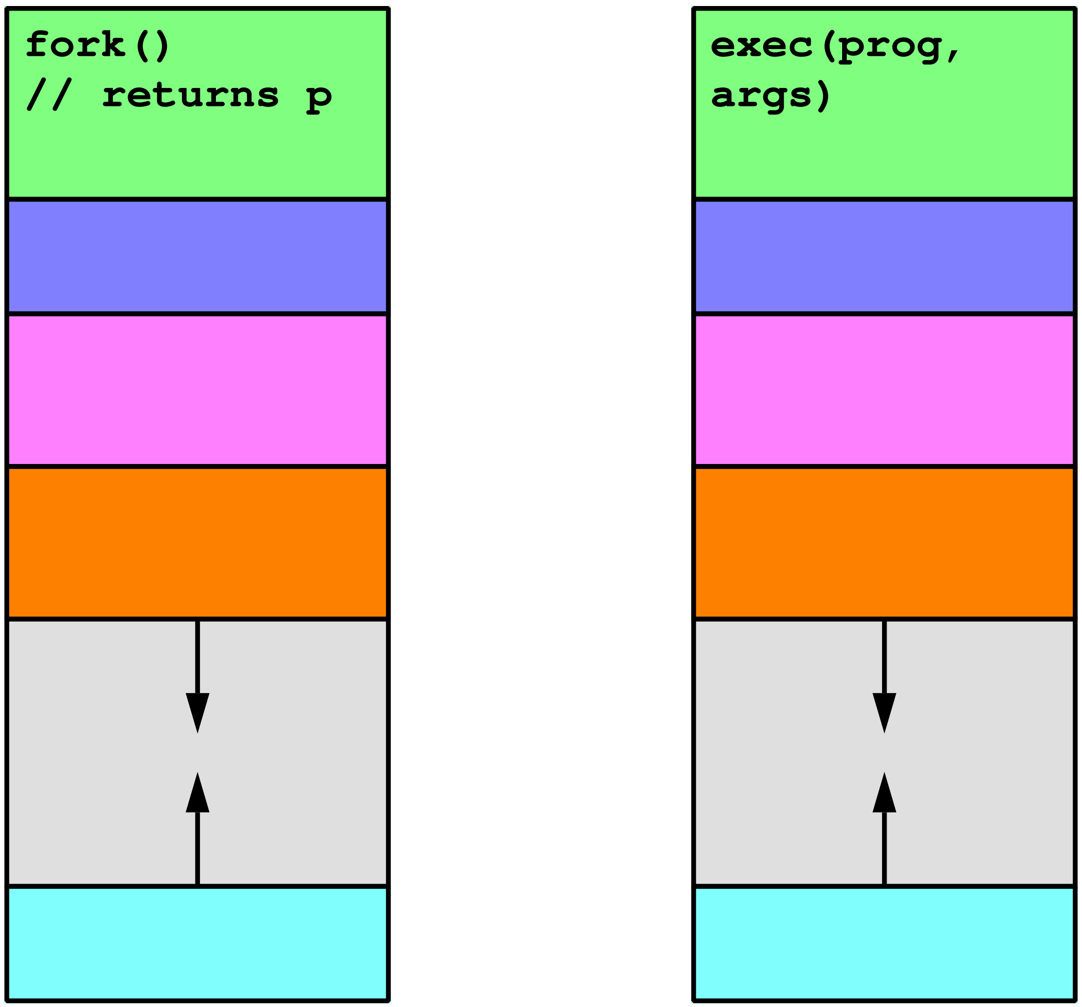
```
int pid;
if ((pid = fork()) == 0) {
  /* we'll discuss what might take place before
     exec is called */
  execl("/home/bc/bin/primes", "primes", "300", 0);
  exit(1);
}
/* parent continues here */
while(pid != wait(0)) /* ignore the return code */
  ;
```

- what does `execl()` do?
  - "man execl" says:

    ```
    int execl(const char *path,
              const char *arg, ...);
    ```

  - isn't "primes" in the 2nd argument kind of redundent?
  - what's up with "..."?
    - this is called *"varargs"* (similar to `printf()`)

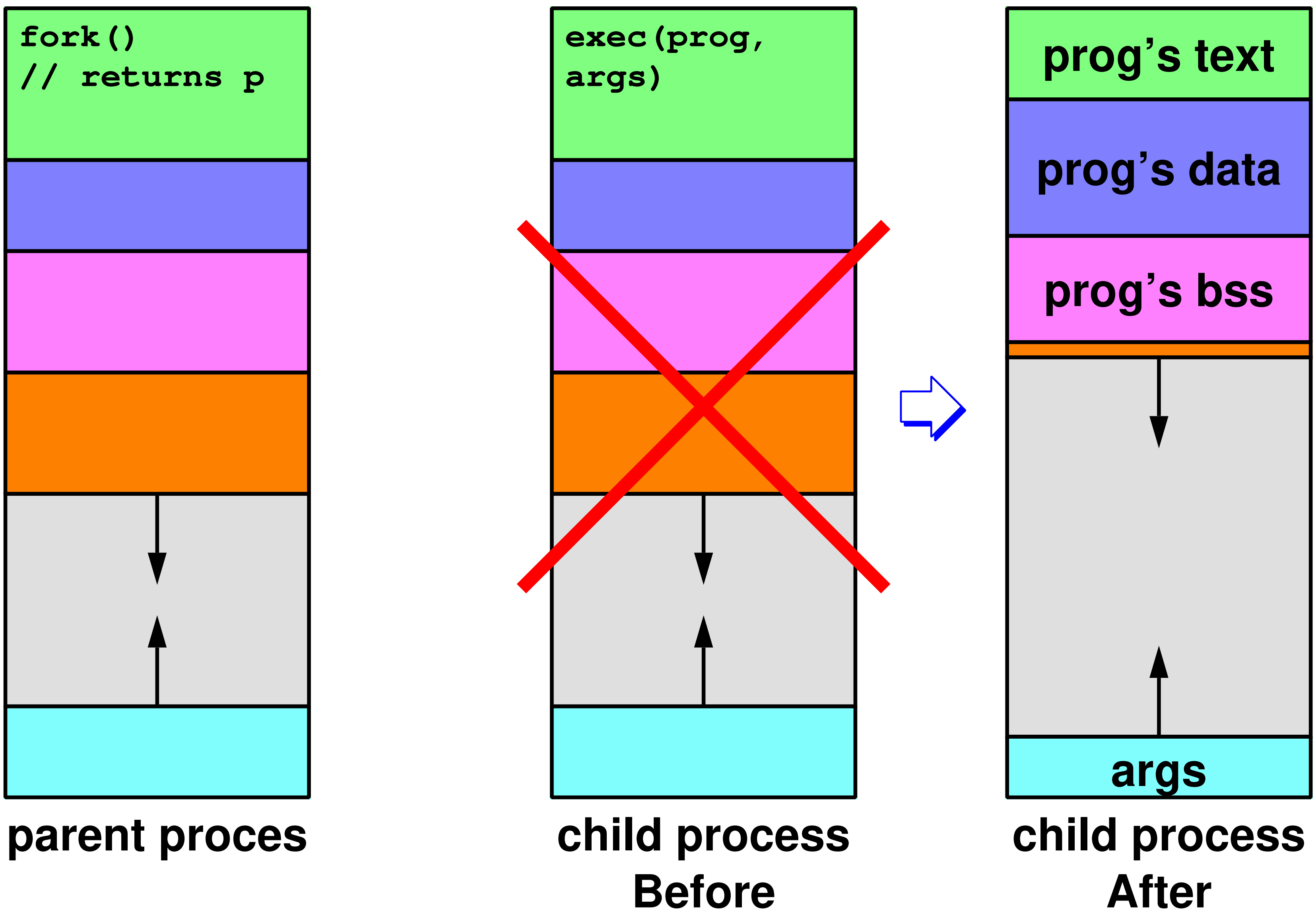
Loading a New Image



parent proces

child process
Before

Loading a New Image



Exec

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes", "primes", "300", 0);
  exit(1);
}
while(pid != wait(0)) /* ignore the return code */
  ;
```

```
% primes 300
```

- Your login shell forks off a child process, load the primes program on top of it, wait for the child to terminate
  - the same code as before
  - `exit(1)` would get called if somehow `execl()` returned
    - if `execl()` is successful, it cannot return since the code is *gone* (i.e., the code segment has been replaced by the code segment of "`primes`")

Put It All Together

Parent (shell)

```
fork()
```

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

Applications

OS

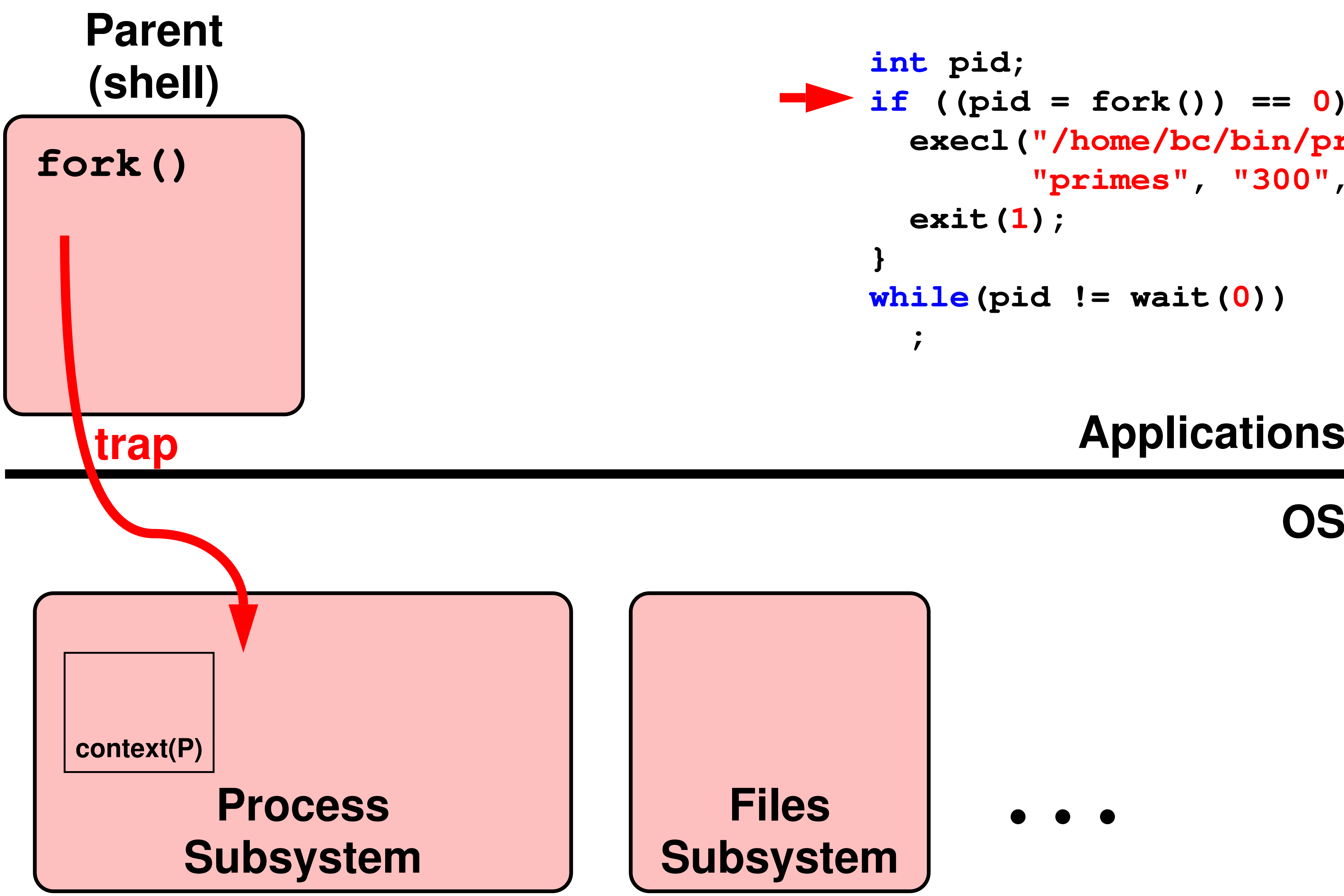
Process Subsystem

Files Subsystem

• • •

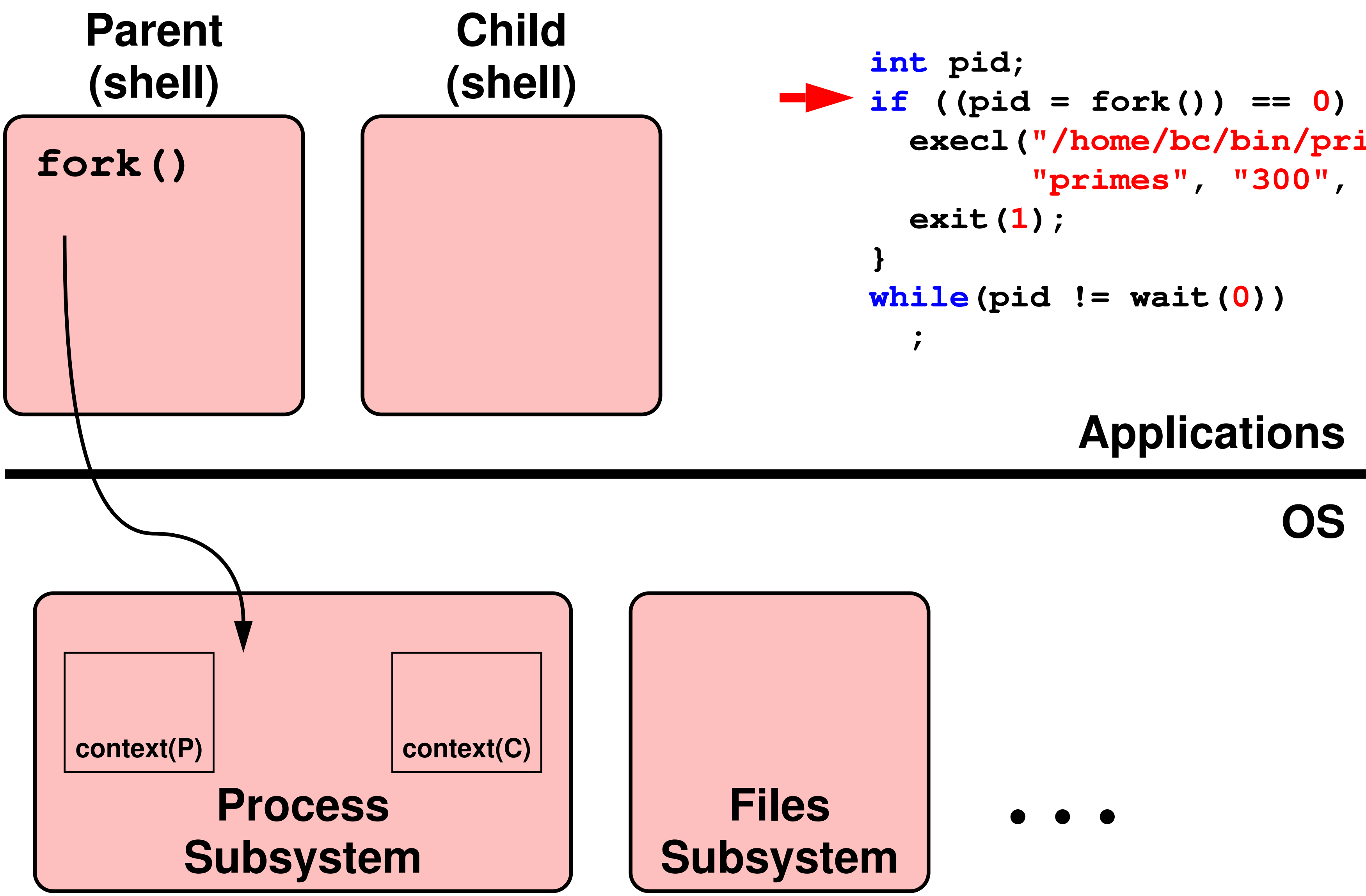
Put It All Together

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```



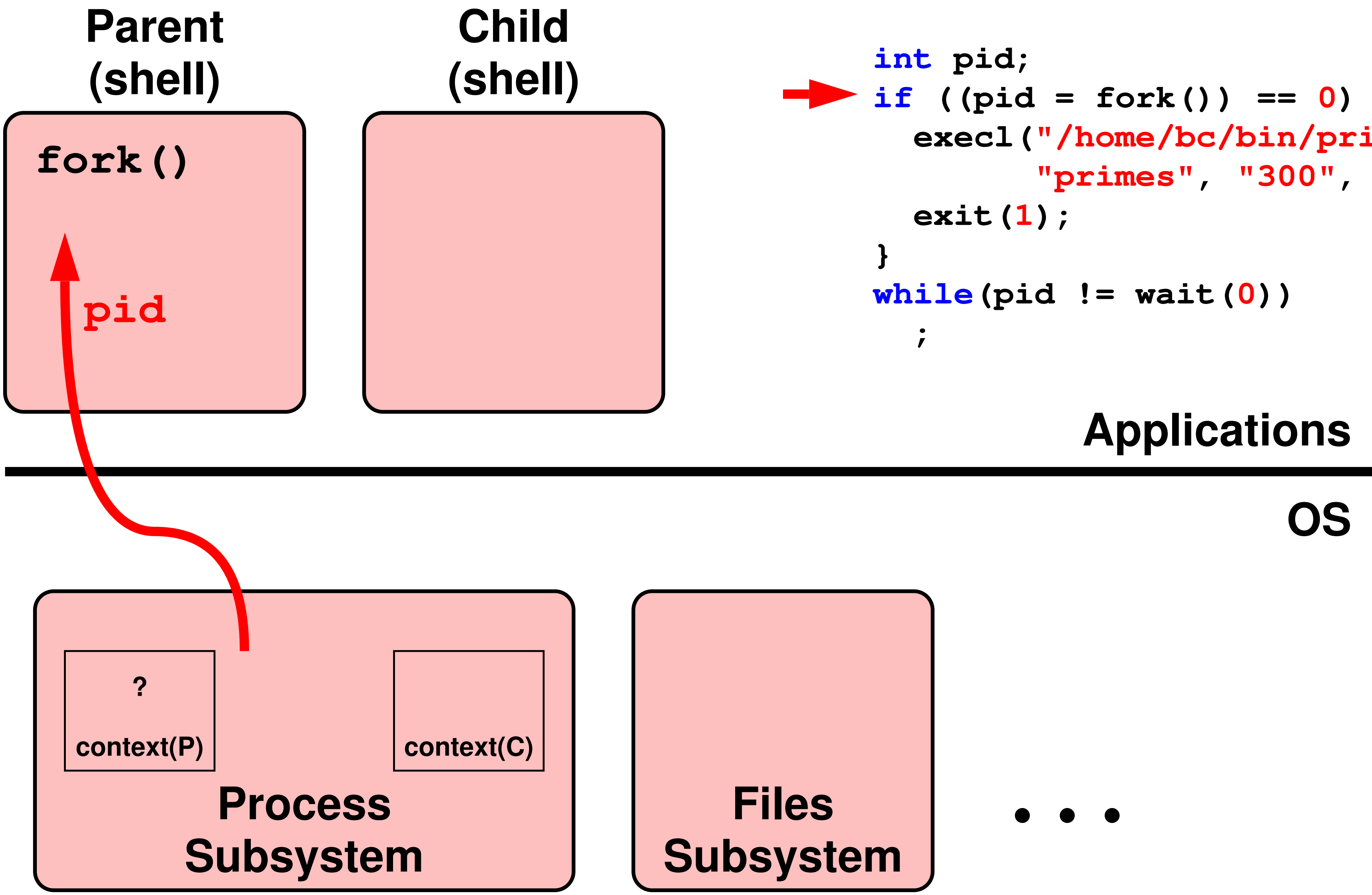
Where do you keep "context"?

Put It All Together



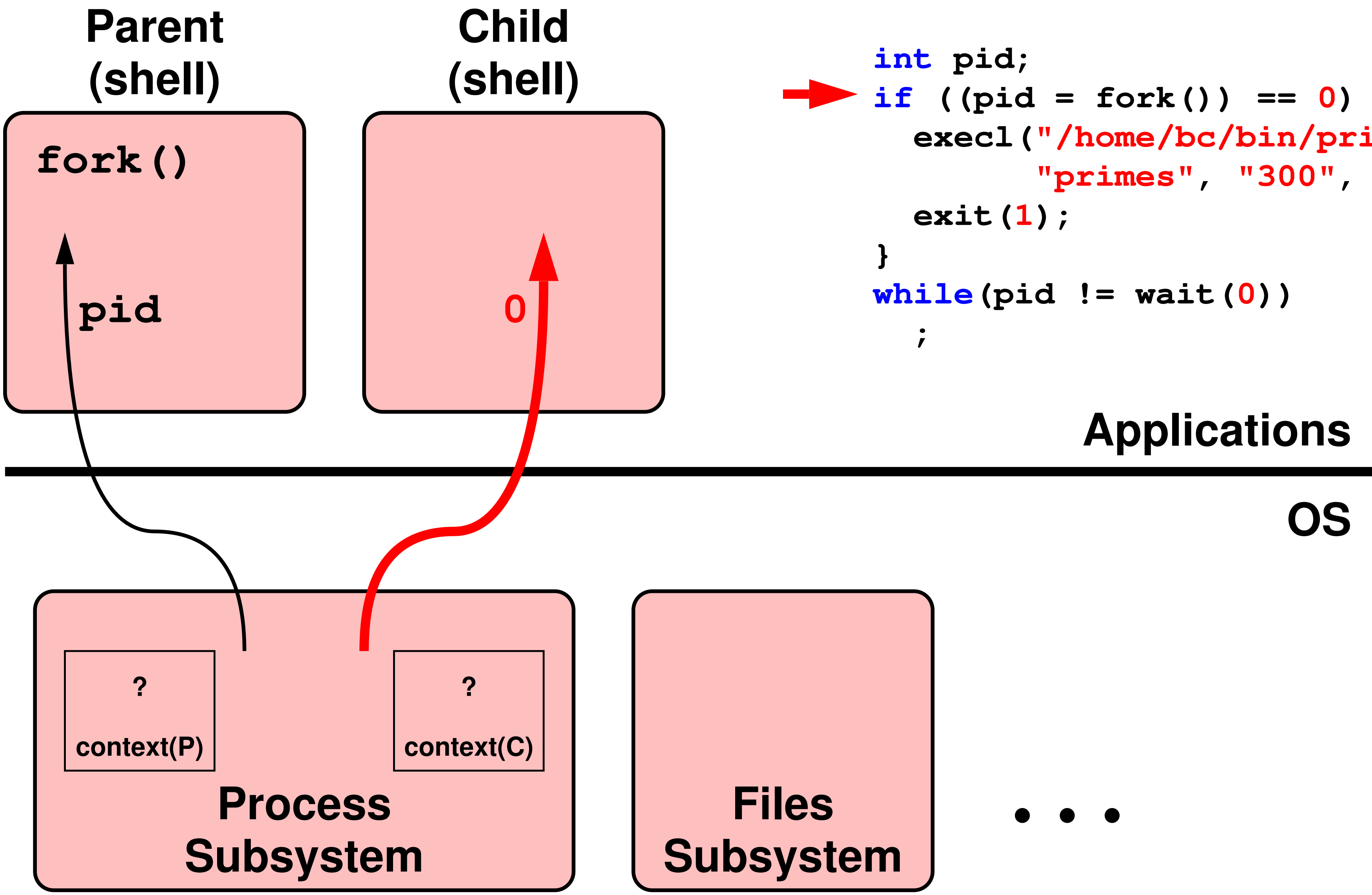

```c
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

Put It All Together



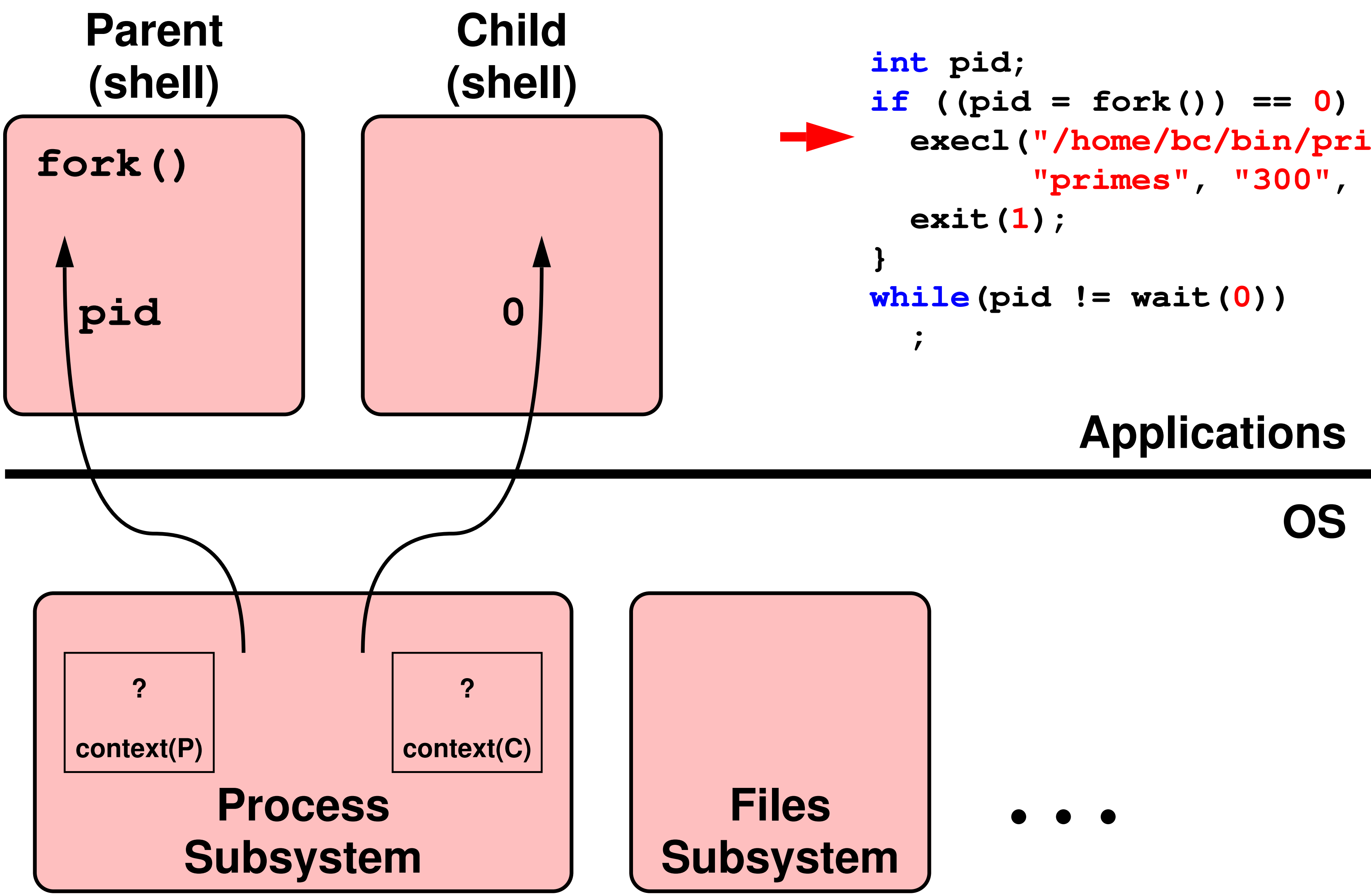

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

Put It All Together



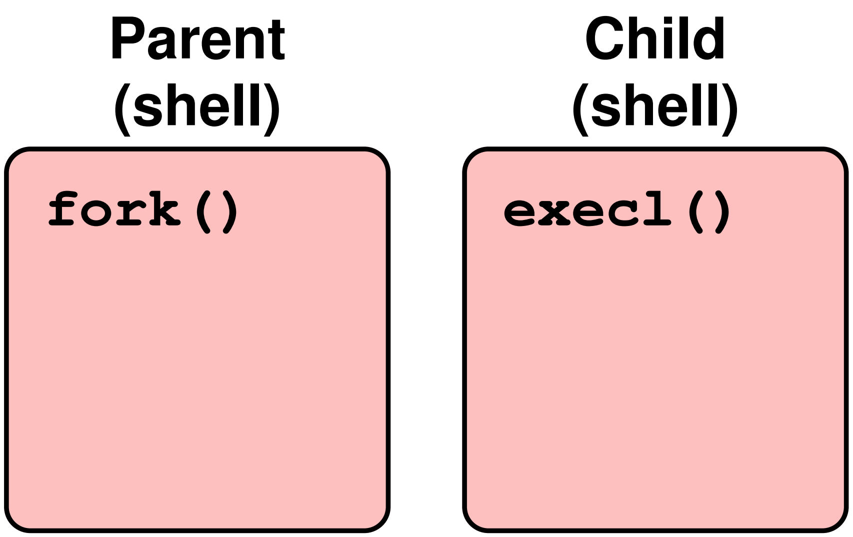

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

Put It All Together

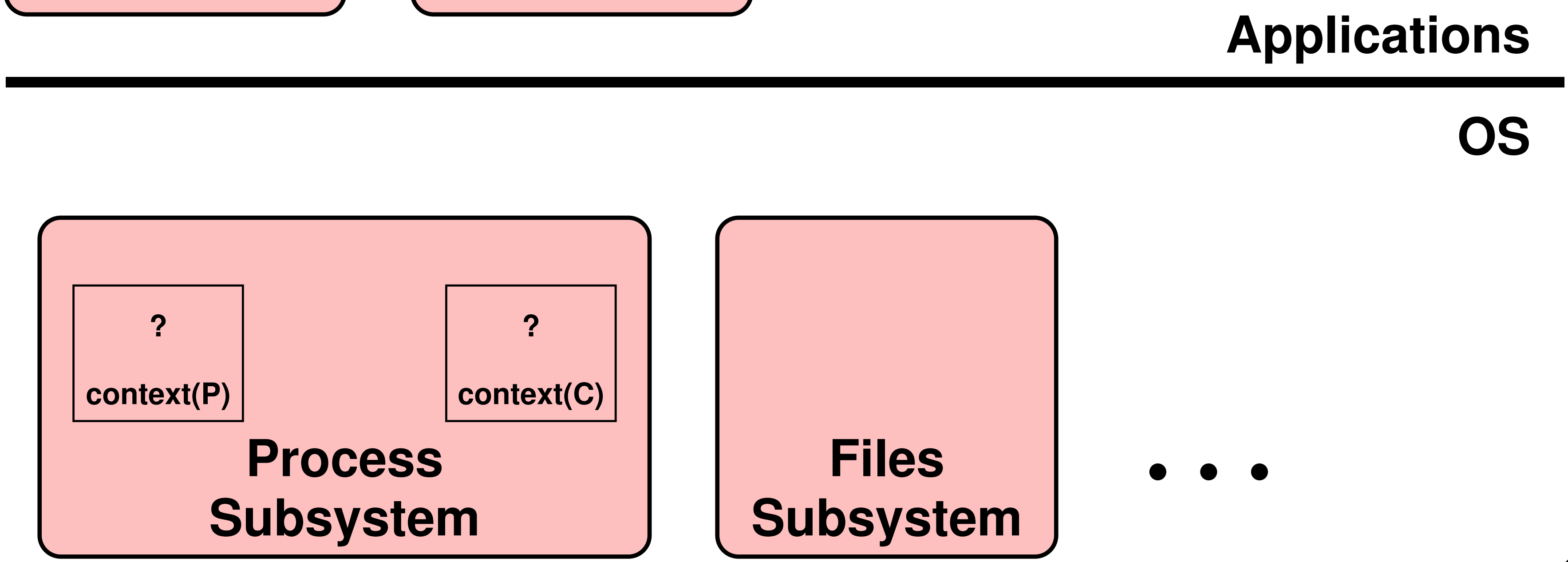



```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

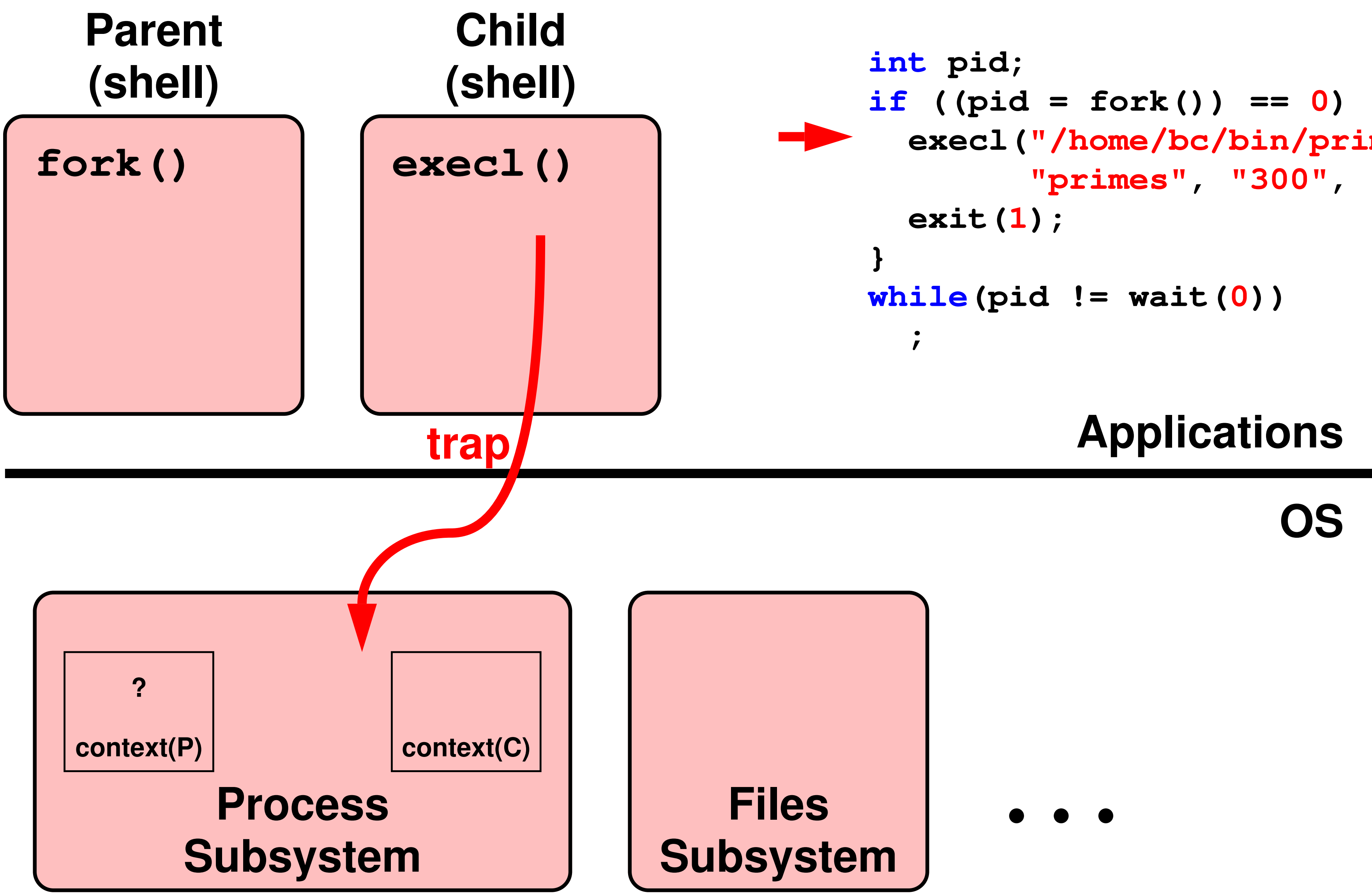
Put It All Together




```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

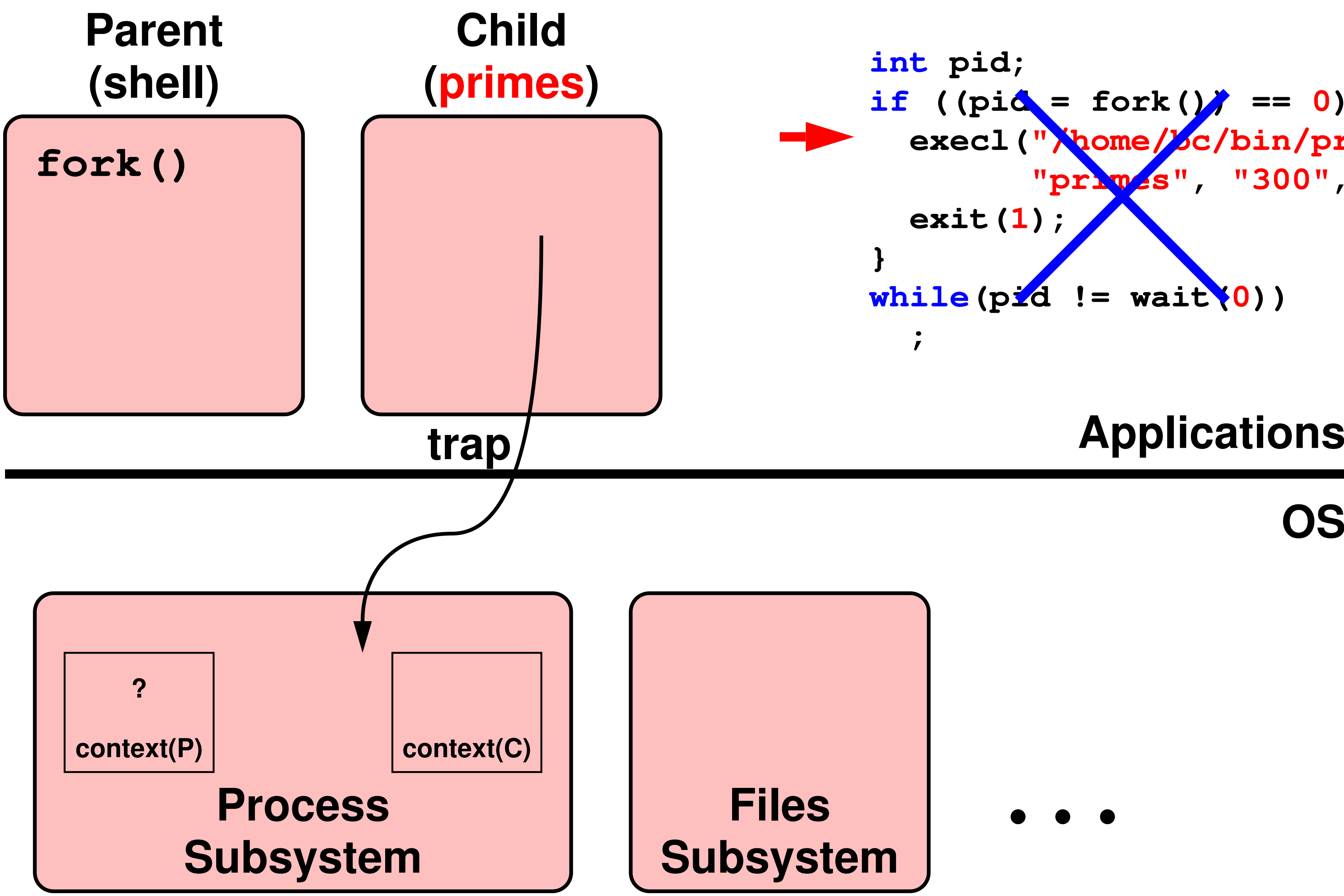


Put It All Together



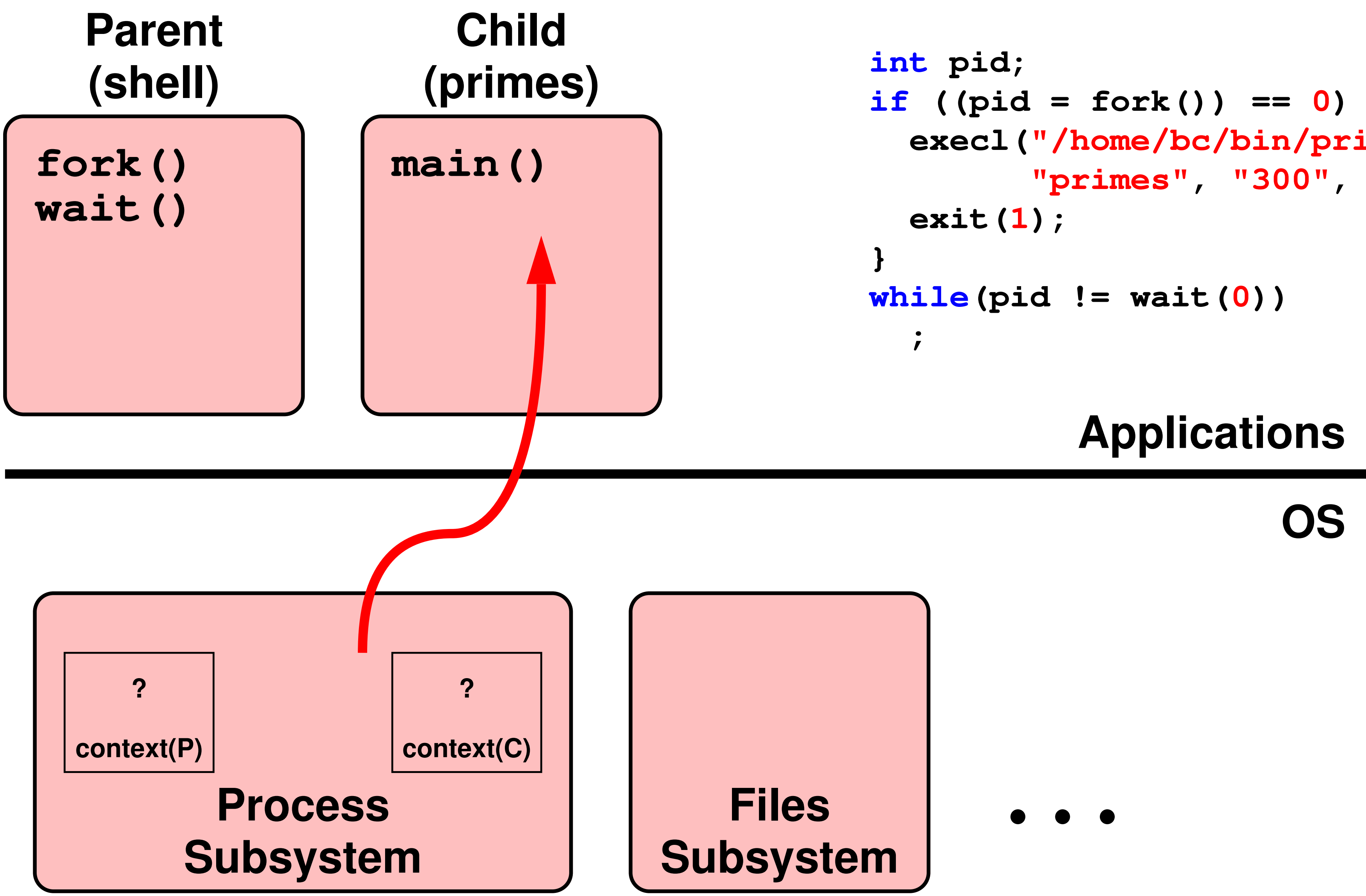

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

Put It All Together




```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

Put It All Together




```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
            "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

Put It All Together

Parent (shell)

```
fork()
wait()
```

Child (primes)

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

Applications

OS

?

context(P)

?

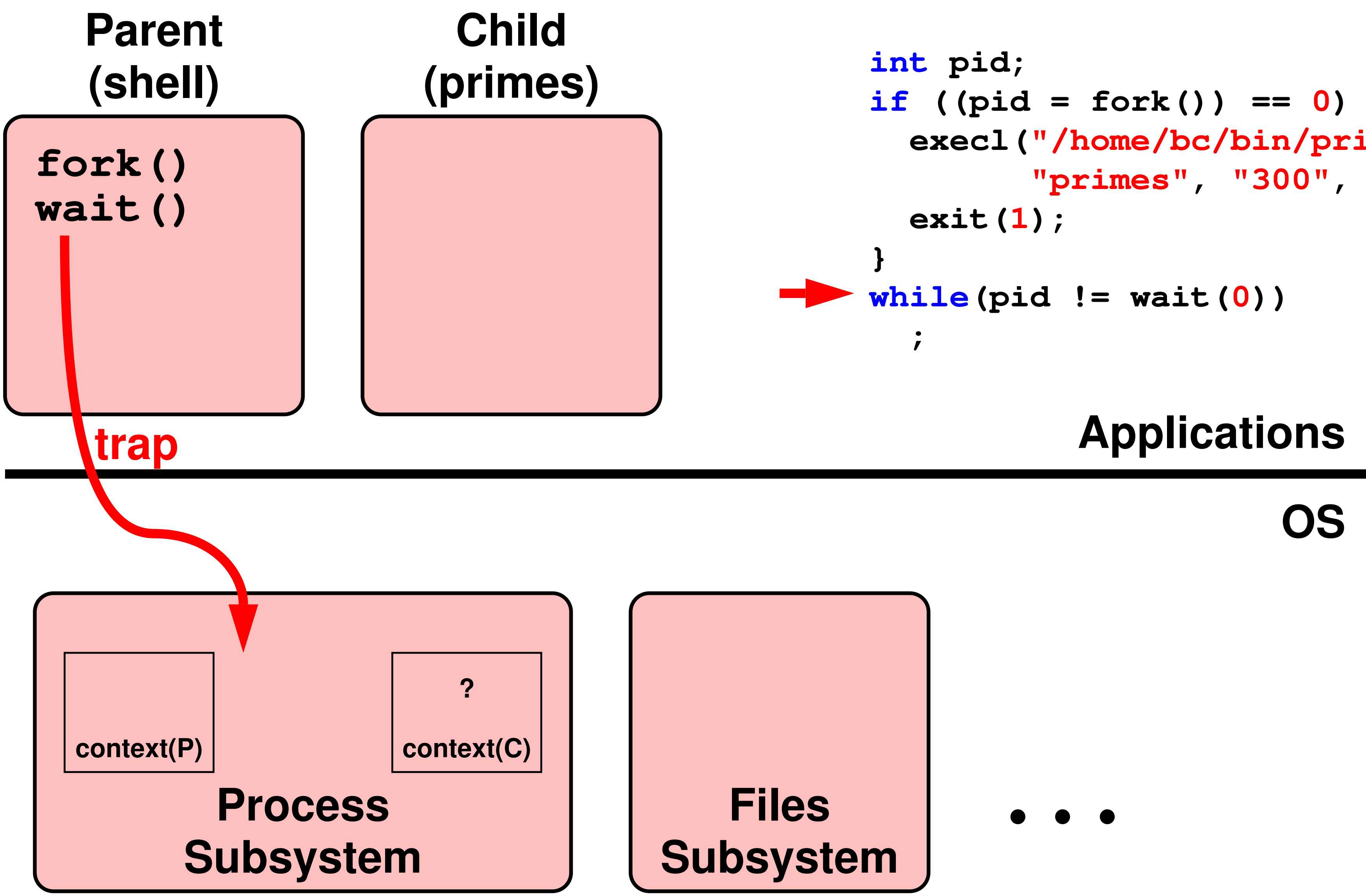
context(C)

Process Subsystem

Files Subsystem

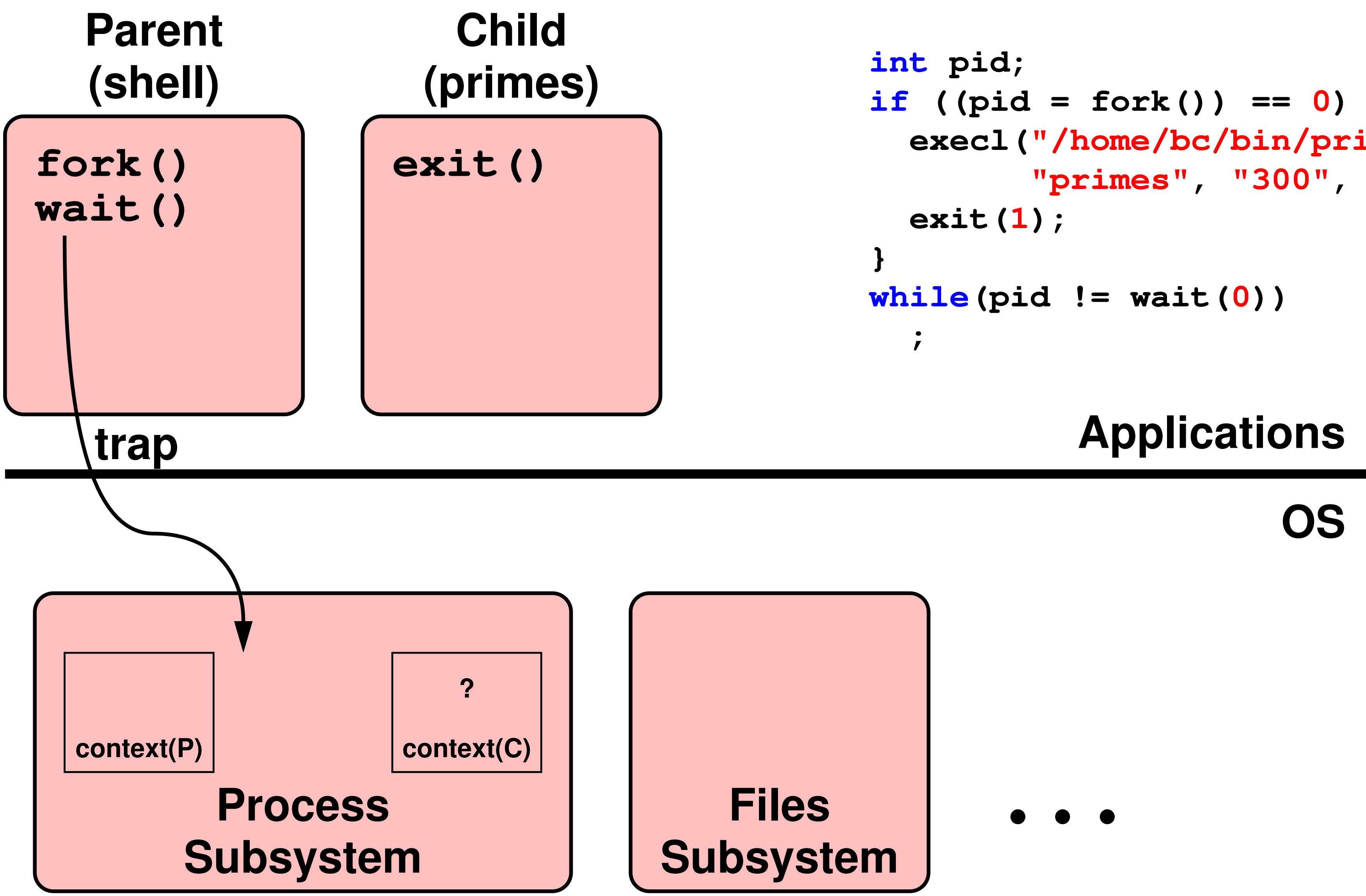
• • •

Put It All Together




```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

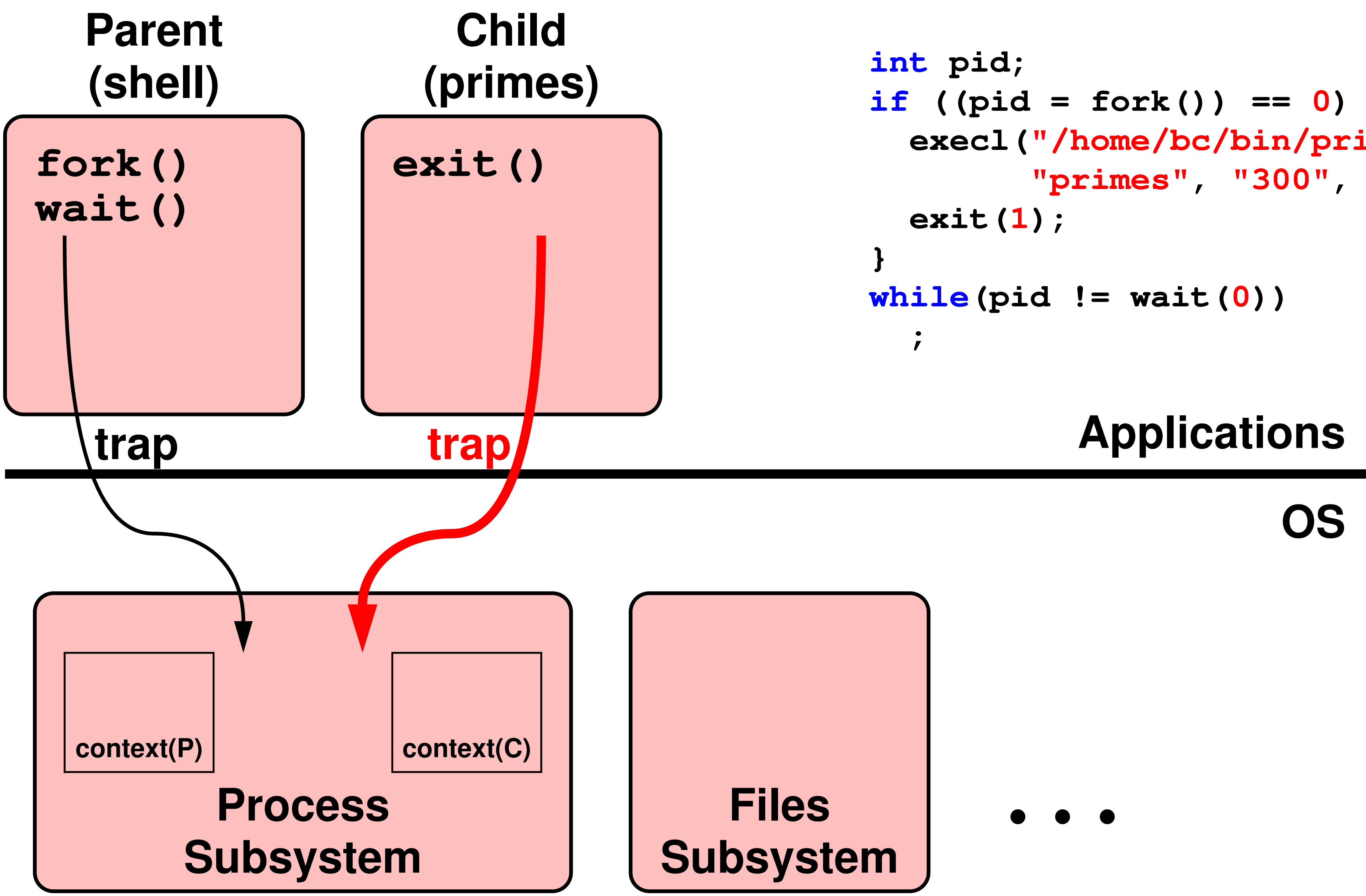
Put It All Together



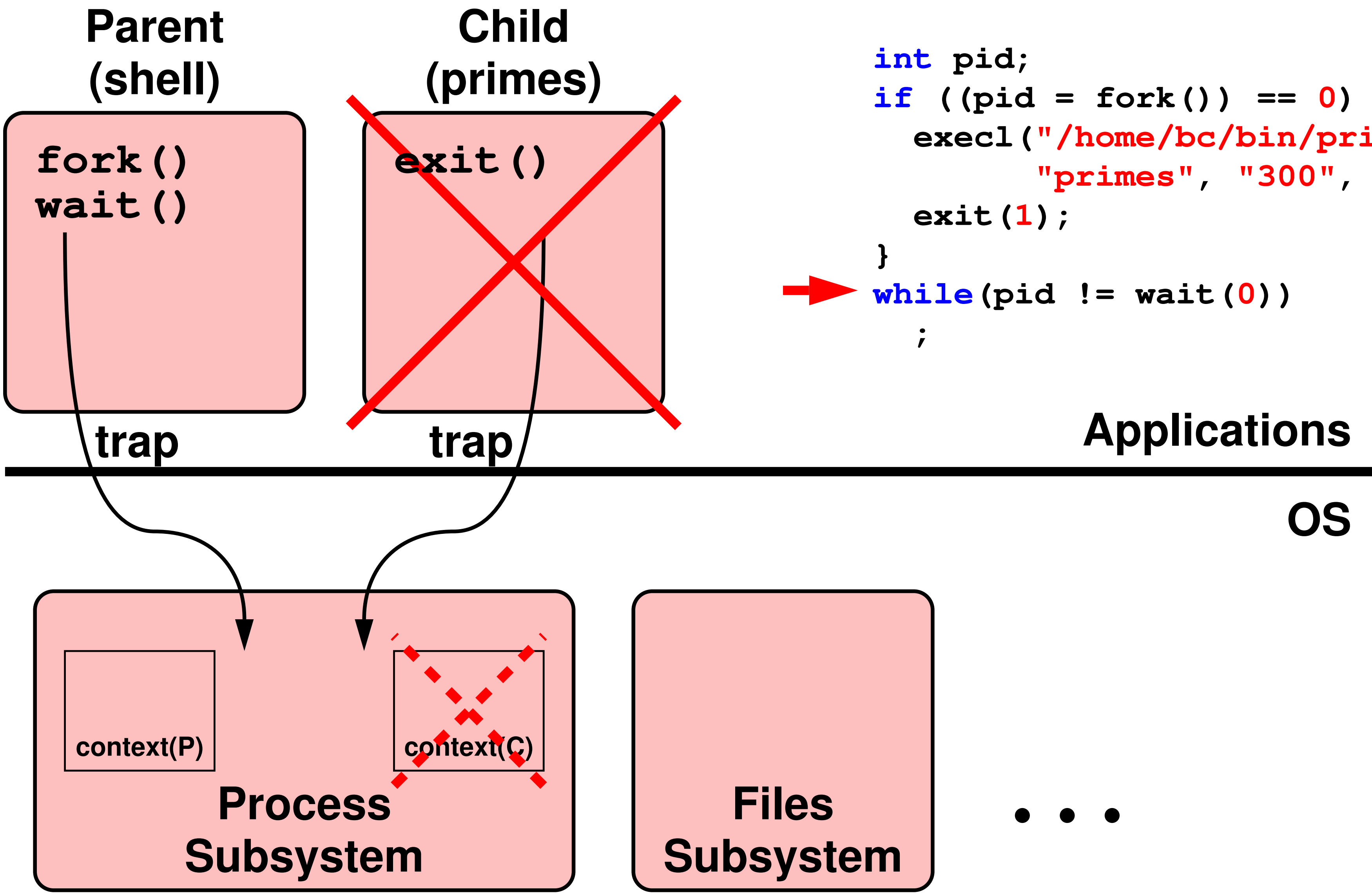

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

Put It All Together

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

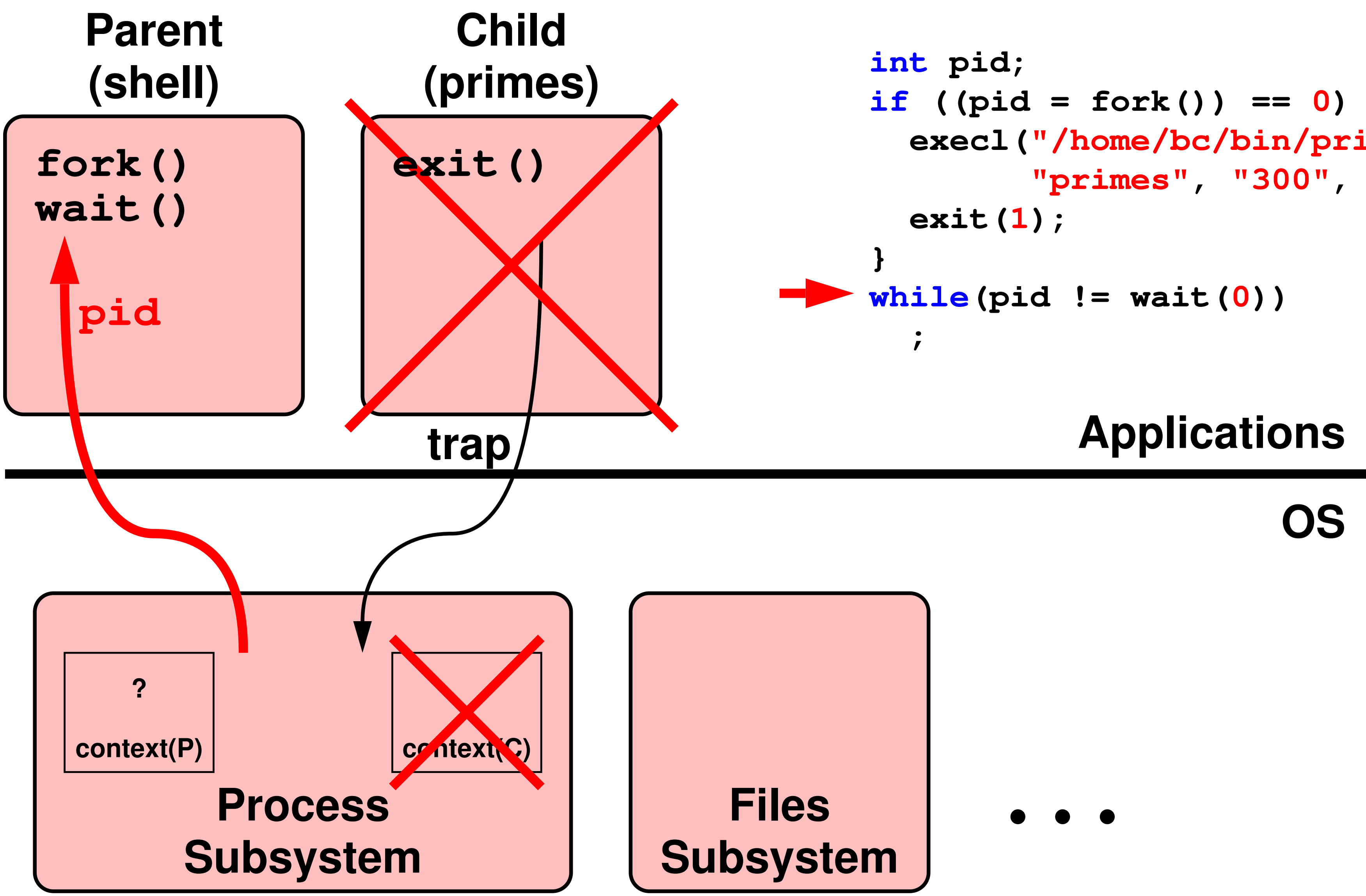


Put It All Together



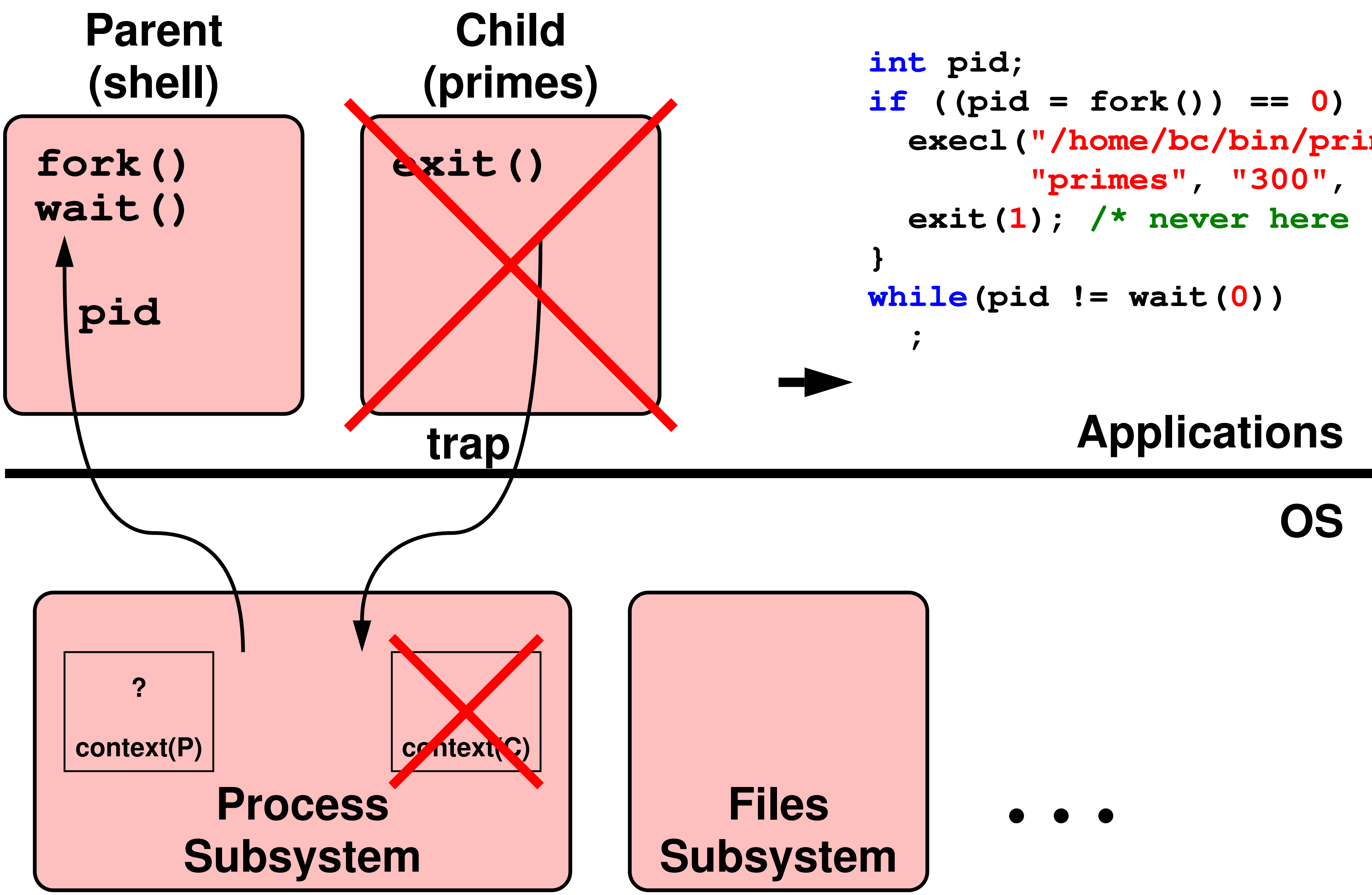

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

Put It All Together




```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

Put It All Together




```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1); /* never here */
}
while(pid != wait(0))
  ;
```