Control Flow

From startup to shutdown, a CPU reads and executes a sequence of instructions.

This sequence is normal **control flow**.

Jumps and calls/returns determine control flow based on **program state**.

```assembly
movq %rax, %rbx
addq %rcx, %rbx
movl (%rbx), %eax
cmpl $0x5, %eax
jne 0x864c22
addq $1, %rax
jmp 0x864a06
```
System State

Changes in *system state*:

- Data arrives from the network
- The user hits Ctrl-C
- A timer expires
- An instruction divides by zero

Need a mechanism for *exceptional control flow*
Kernel vs. User Code

When you turn on a processor, instructions can do anything: the processor starts in *privileged mode*

```
mov 42, 0x75462
```
In privileged mode, the kernel can change the way that *virtual addresses* are mapped to physical memory.

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x75xxx</td>
<td>0x36xxx</td>
</tr>
</tbody>
</table>

```
mov 43, 0x75462
⇒
0x75xxx
0x36xxx
```

So, the kernel can hide memory from unprivileged user code.

*but, before doing that...*
Kernel vs. User Code

Special register **IDTR** points to memory (not accessible to user code) for a table of functions to handle **exceptions**:

This is the **exception table**

a.k.a. the **interrupt vector**
Kernel vs. User Code

Special register \textbf{IDTR} points to memory (not accessible to user code) for a table of functions to handle \textit{exceptions}:

Call exception handler: ignore address remappings and switch back to privileged mode

\textbf{Control the table} ⇒ control the way back to privileged mode
Kernel vs. User Code

Special register **IDTR** points to memory (not accessible to user code) for a table of functions to handle **exceptions**:

\[
\text{IDTR} = \begin{array}{c}
\text{on\_divide\_by\_zero} \\
\text{...} \\
\text{...} \\
\text{on\_syscall}
\end{array}
\]

**int** \( k \)

Trigger \( k \) exception

\( k = 0x80 \) means “system call”

\[
\begin{align*}
\text{mov} &\quad $0x2, \%eax \\
\text{int} &\quad $0x80
\end{align*}
\]
Kernel vs. User Code

Special register **IDTR** points to memory (not accessible to user code) for a table of functions to handle **exceptions**: 

```
IDTR = ... on_divide_by_zero ...
    ... on_syscall ...
```

**syscall**

Same idea as `int $0x80`, but faster

```
mov $0x2,%eax
syscall
```
Exception Handling

User code

... $I_{n-1}$ $I_{n}$ $I_{current}$ $I_{next}$ $I_{n+3}$ $I_{n+4}$ ...

Exception Handling

User code

Kernel code

...  
$I_{n-1}$
$I_n$
$I_{current}$
$I_{next}$
$I_{n+3}$
$I_{n+4}$
...

exception $k$
Exception Handling

User code

... 
$I_{n-1}$
$I_n$
$I_{current}$
$I_{next}$
$I_{n+3}$
$I_{n+4}$
...

Kernel code

exception $k$

exception handler for $k$
Exception Handling

User code

... $I_{n-1}$ $I_n$ $I_{\text{current}}$ $I_{\text{next}}$ $I_{n+3}$ $I_{n+4}$ ...

Kernel code

exception $k$ exception handler for $k$ abort

Exception Handling

User code

... 
$I_{n-1}$ 
$I_n$ 
$I_{\text{current}}$ 
$I_{\text{next}}$ 
$I_{n+3}$ 
$I_{n+4}$ 
... 

Kernel code

retry at $I_{\text{current}}$

exception $k$

exception handler for $k$

abort

Exception Handling

User code

Kernel code

...  
$I_{n-1}$  
$I_n$  
$I_{current}$  
$I_{next}$  
$I_{n+3}$  
$I_{n+4}$  
...

retry at $I_{current}$

exception $k$

resume at $I_{next}$

abort

exception handler for $k$
Four Kinds of Exceptions

User
Kernel

CPU
other hardware
Four Kinds of Exceptions

**interrupt** — from hardware: keyboard, network packet, ...

- *asynchronous* with respect to the program
- handled by kernel, which then *resumes* program
Four Kinds of Exceptions

interrupt asynch resume

User
Kernel

CPU
other hardware
Four Kinds of Exceptions

**trap** — from program: system call, breakpoint, ...

- *synchronous* and *intentional*
- handled by kernel, which then *resumes* program
Four Kinds of Exceptions

**fault** — by program: bad memory reference, ...

- *synchronous* and usually *unintentional*
- handled by kernel, which may *retry* or *abort*
  ...maybe with program help

![Diagram showing the relationship between User, Kernel, CPU, and trap events]
Four Kinds of Exceptions

**abort** — hardware errors and such

- *synchronous* and *unintentional*
- kernel takes emergency measures to **abort**
Four Kinds of Exceptions

- **interrupt**: asynchronous
- **trap**: intentional
- **fault**: unintentional
- **abort**: unintentional

User → Kernel

CPU

interrupt

trap

fault

abort

other hardware
Exceptions explain how an OS can control your code:

- External **interrupts** ⇒ kernel can handle network, etc.
- Timer **interrupt** ⇒ kernel gets control often enough
- System calls via **trap** ⇒ kernel as more privileged
- Errors as **faults** ⇒ kernel can take over
Switching User Code

User

Kernel

interrupt

CPU
Switching User Code

User

Kernel

CPU
Switching User Code

User

Kernel

CPU
Switching User Code
Switching User Code
Switching User Code
Switching user code is a context switch
Switching User Code

Program
Memory

Program
Memory
Kernel

CPU
Switching User Code
Process

A **process** is a running **instance** of a program

Each process gets:

- **local control flow**
  a program seems to have the whole CPU

- **private address space**
  a program seems to have all of memory
Process

A *process* is a running *instance* of a program.

Each process gets:

- **local control flow**
  - a program seems to have the whole CPU

- **private address space**
  - a program seems to have all of memory

*Computer Systems: A Programmer’s Perspective, Third Edition*
Multiprocessing: The Illusion

![Diagram showing memory and CPU sections with stack, heap, code, and registers.](image-url)
Multiprocessing: The Reality (Single Core)
Multiprocessing: The Reality (Single Core)
Multiprocessing: The Reality (Single Core)
Multiprocessing: The Reality (Single Core)
Multiprocessing: The Reality (Multicore)
### Multiprocessing Concurrency

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>user code</td>
<td>kernel code } context switch</td>
</tr>
<tr>
<td>kernel code</td>
<td>user code</td>
</tr>
<tr>
<td>user code</td>
<td>kernel code } context switch</td>
</tr>
<tr>
<td>kernel code</td>
<td>user code</td>
</tr>
<tr>
<td>user code</td>
<td>kernel code } context switch</td>
</tr>
<tr>
<td>kernel code</td>
<td>user code</td>
</tr>
<tr>
<td>user code</td>
<td>kernel code } context switch</td>
</tr>
</tbody>
</table>
Multiprocessing Concurrency
#top

```
File  Edit  View  Search  Terminal  Help

top - 06:54:47 up 8 days, 8:04, 2 users, load average: 0.29, 0.09, 0.08
Tasks: 177 total, 2 running, 175 sleeping, 0 stopped, 0 zombie
%Cpu(s): 11.6 us, 0.7 sy, 0.0 ni, 87.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.
KiB Mem: 1560592 total, 150756 free, 703452 used, 706384 buff/cache
KiB Swap: 1257468 total, 992736 free, 264732 used, 670244 avail Mem

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PR</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>S</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME+</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>15945</td>
<td>mflatt</td>
<td>20</td>
<td>0</td>
<td>1549092</td>
<td>259756</td>
<td>14532</td>
<td>S</td>
<td>12.3</td>
<td>16.6</td>
<td>201:29.76</td>
<td>gnome-s+</td>
</tr>
<tr>
<td>7841</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>255548</td>
<td>49036</td>
<td>3508</td>
<td>S</td>
<td>3.7</td>
<td>3.1</td>
<td>160:10.31</td>
<td>X</td>
</tr>
<tr>
<td>457</td>
<td>mflatt</td>
<td>20</td>
<td>0</td>
<td>1003284</td>
<td>170764</td>
<td>51552</td>
<td>S</td>
<td>3.3</td>
<td>10.9</td>
<td>0:05.58</td>
<td>firefox</td>
</tr>
<tr>
<td>384</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.3</td>
<td>0.0</td>
<td>2:54.44</td>
<td>xfsaidl+</td>
</tr>
<tr>
<td>9026</td>
<td>mflatt</td>
<td>20</td>
<td>0</td>
<td>473224</td>
<td>9960</td>
<td>952</td>
<td>S</td>
<td>0.3</td>
<td>0.6</td>
<td>12:45.92</td>
<td>ibus-da+</td>
</tr>
<tr>
<td>9039</td>
<td>mflatt</td>
<td>20</td>
<td>0</td>
<td>558012</td>
<td>13308</td>
<td>5628</td>
<td>S</td>
<td>0.3</td>
<td>0.9</td>
<td>2:37.71</td>
<td>gnome-t+</td>
</tr>
<tr>
<td>9044</td>
<td>mflatt</td>
<td>20</td>
<td>0</td>
<td>452392</td>
<td>2020</td>
<td>1048</td>
<td>S</td>
<td>0.3</td>
<td>0.1</td>
<td>10:38.84</td>
<td>ibus-x11</td>
</tr>
<tr>
<td>9172</td>
<td>mflatt</td>
<td>20</td>
<td>0</td>
<td>581396</td>
<td>10624</td>
<td>1428</td>
<td>S</td>
<td>0.3</td>
<td>0.7</td>
<td>1:43.52</td>
<td>caribou</td>
</tr>
<tr>
<td>20291</td>
<td>mflatt</td>
<td>20</td>
<td>0</td>
<td>585084</td>
<td>43416</td>
<td>17460</td>
<td>S</td>
<td>0.3</td>
<td>2.8</td>
<td>0:42.26</td>
<td>emacs</td>
</tr>
<tr>
<td>1</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>126516</td>
<td>4908</td>
<td>2404</td>
<td>S</td>
<td>0.0</td>
<td>0.3</td>
<td>0:38.40</td>
<td>systemd</td>
</tr>
<tr>
<td>2</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.49</td>
<td>kthreadd</td>
</tr>
<tr>
<td>3</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:03.07</td>
<td>ksoftirq+</td>
</tr>
<tr>
<td>7</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>migratid+</td>
</tr>
<tr>
<td>8</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>rcu_bh</td>
</tr>
<tr>
<td>9</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>rcu_ob/0</td>
</tr>
</tbody>
</table>
```
A CPU-Wasting Program

```c
int main() {
    while (1) {
    }
}
```
ps and kill

List some processes:

$ ps

List all processes started by you:

$ ps x

List all processes:

$ ps ax
Interrupt a process:

```
$ kill id
```
An Uncooperative CPU-Wasting Program

```c
#include <signal.h>

int main() {
    signal(SIGINT, SIG_IGN);
    signal(SIGTERM, SIG_IGN);
    while (1) { }
}
```
Interrupt an uncooperative program:

$ kill -SIGKILL \textit{pid}

or

$ kill -9 \textit{pid}
getpid

```c
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
```

Gets the current process’s ID as an integer
getppid

```
#include <sys/types.h>
#include <unistd.h>

pid_t getppid(void);
```

Gets the ID of the process that started the current process
Getting A Process ID

In /usr/lib64/libc.so.6:

\texttt{<getppid>:
  \texttt{mov \ $0x6e, \%eax}
  \texttt{syscall}
  \texttt{retq}}
The C Library vs. System Calls

Opening a file in **portable C**:  

```c
FILE *f = fopen("data.txt", "r");
```

Opening a file in **Unix**:

```c
int f = open("data.txt", O_RDONLY);
```

`man fopen` ⇒ `FOPEN(3)`  
(3) means “C library”

`man open` ⇒ `OPEN(2)`  
(2) means “system call”
int f = open("nosuchfile.txt", O_RDONLY);

No exception... just a −1 value for f
System Calls and Error

Most system calls return an integer result
Most report an error as a -1 result
The errno variable provides details

```c
int f = open("nosuchfile.txt", O_RDONLY);
if (f == -1) {
    /* Handle error */
    fprintf(stderr, "open failed (%s)",
            strerror(errno));
    exit(1);
}
```

This is a pain...
Syscall Wrapper for Errors

Slightly simplified `open` implementation:

```assembly
<open>:
  e82a9:  mov    $0x2,%eax
  e82ae:  syscall
  e82b0:  cmp    $0xffffffffffffff001,%rax
  e82b6:  jae    e82e9   # jump if in error range
  e82b8:  retq
  e82e9:  mov    0x2d2b78(%rip),%rcx     # &errno
  e82f0:  neg    %eax
  e82f2:  mov    %eax,(%rcx)          # set errno
  e82f5:  or     $0xffffffffffffffff,%rax
  e82f9:  retq
```
Syscall Wrapper for Errors

Slightly simplified `open` implementation:

```assembly
<open>:
    e82a9:  mov    $0x2,%eax
    e82ae:  syscall
    e82b0:  cmp    $0xfffffffffffff001,%rax
    e82b6:  jae    e82e9   # jump if in error range
    e82b8:  retq
    e82e9:  mov    0x2d2b78(%rip),%rcx     # &errno
    e82f0:  neg    %eax
    e82f2:  mov    %eax,(%rcx)          # set errno
    e82f5:  or     $0xfffffffffffffffefa,0x2
    e82f9:  retq
```

0x2 means `open`
Syscall Wrapper for Errors

Slightly simplified `open` implementation:

```assembly
<open>:
e82a9:  mov    $0x2,%eax
e82ae:  syscall
e82b0:  cmp    $0xfffffffffffff001,%rax
e82b6:  jae    e82e9     # jump if in error range
e82b8:  retq
e82e9:  mov    0x2d2b78(%rip),%rcx     # &errno
e82f0:  neg    %eax
e82f2:  mov    %eax,(%rcx)          # set errno
e82f5:  or     $0xffffffffffffffff,%rax
e82f9:  retq
```

-1 to -4096 means an error
Syscall Wrapper for Errors

Slightly simplified `open` implementation:

```assembly
<open>:
  e82a9:  mov    $0x2,%eax
  e82ae:  syscall
  e82b0:  cmp    $0xffffffffffff001,%rax
  e82b6:  jae    e82e9   # jump if in error range
  e82b8:  retq
  e82e9:  mov    0x2d2b78(%rip),%rcx     # &errno
  e82f0:  neg    %eax
  e82f2:  mov    %eax,(%rcx)          # set errno
  e82f5:  or     $0xfffffffffffffff,%rax
  e82f9:  retq
```

Syscall Wrapper for Errors

Slightly simplified `open` implementation:

```
<open>:
    e82a9:  mov    $0x2,%eax
    e82ae:  syscall
    e82b0:  cmp    $0xfffffffffffff001,%rax
    e82b6:  jae    e82e9   # jump if in error range
    e82b8:  retq
    e82e9:  mov    0x2d2b78(%rip),%rcx     # &errno
    e82f0:  neg    %eax
    e82f2:  mov    %eax,(%rcx)          # set errno
    e82f5:  or     $0xffffffffffffffff,%rax
    e82f9:  retq
```

negate result as `errno`
Syscall Wrapper for Errors

Slightly simplified `open` implementation:

```
<open>:
  e82a9:  mov   $0x2,%eax
  e82ae:  syscall
  e82b0:  cmp   $0xfffffffffffffffff001,%rax
  e82b6:  jae   e82e9  # jump if in error range
  e82b8:  retq
  e82e9:  mov   0x2d2b78(%rip),%rcx     # &errno
  e82f0:  neg   %eax
  e82f2:  mov   %eax,(%rcx)          # set errno
  e82f5:  or    $0xfffffffffffffff001,%rax
  e82f9:  retq
```

return -1
Textbook Wrapper for Errors

More help from `csapp.h` and `csapp.c`:

```c
void unix_error(char *msg) {
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(0);
}

int Open(const char *pathname, int flags, mode_t mode) {
    int rc;
    if ((rc = open(pathname, flags, mode)) < 0)
        unix_error("Open error");
    return rc;
}
```
Creating a New Process

The system call that you’d expect:

```c
int newprocess(char *prog, int argc, char **argv);
```

Create a new process with a given program

If Program₁ starts Program₂:

![Diagram showing the process of Program₁ starting Program₂]
Creating a New Process

The system calls provided by Unix:

```c
#include <unistd.h>

int fork();
int execve(char *prog, char **argv, char **env);
```

**fork** creates a *copy* of the current process

**execve** *replaces* the current process

![Diagram showing the effect of fork and execve on program and memory]
Creating a New Process

The system calls provided by Unix:

```c
int fork();
int execve(char *prog, char **argv, char **env);
```

newprocess = fork + execve
Fork

```c
#include <unistd.h>

pid_t fork(void);
```

Creates a new process as a copy of the current one, but:

- Copy has a different PID
- Returns that PID to the original, `parent` process
- Returns 0 to the new, `child` process

**Called once, returns twice!**
#include "csapp.h"

int main() {
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        printf("child : x=%d\n", ++x);
    } else {
        /* Parent */
        printf("parent: x=%d\n", --x);
    }

    return 0;
}

• Separate copies of \texttt{x}

• Order of \texttt{printf} unspecified
Process Graphs

We can reason about concurrency with a **process graph**

```c
int main() {
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        printf("child: x=%d\n", ++x);
    } else {
        /* Parent */
        printf("parent: x=%d\n", --x);
    }

    return 0;
}
```

Each node ● is an externally visible action

Edges can be annotated with internal state changes

A *topological sort* of the graph is a possible ordering of events

Ordering by Process Graph

child: \( x = 2 \)

parent: \( x = 0 \)

possible order: \( a \ b \ e \ c \ f \ d \)

impossible order: \( a \ b \ f \ c \ e \ d \)
Consecutive Forks

```c
int main() {
    printf("L0\n");
    Fork();
    printf("L1\n");
    Fork();
    printf("Bye\n");
    return 0;
}
```

Possible output:  Impossible output:
L0               L0
L1               Bye
Bye              L1
Bye              Bye
Bye              Bye
Nested Forks in Parent

int main() {
    printf("L0\n");
    if (Fork() != 0) {
        printf("L1\n");
        if (Fork() != 0)
            printf("L2\n");
    }
    printf("Bye\n");
}

Possible output:  Impossible output:
L0               L0
L1               L1
Bye              Bye
Bye              Bye
Bye              Bye
Bye              Bye

Nested Forks in Children

```
int main() {
    printf("L0\n");
    if (Fork() == 0) {
        printf("L1\n");
        if (Fork() == 0)
            printf("L2\n");
    }
    printf("Bye\n");
}
```

Possible output:  
L0  
Bye  
L1  
L2  
Bye  
Bye

Impossible output:  
L0  
Bye  
L1  
Bye  
Bye  
L2