System Programming: Signals Handling

Raymond Namyst
Dept. of Computer Science
University of Bordeaux, France

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

The concept of signal

- Signals are notifications sent to processes
  - Signals are just numbers (no additional information attached)

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGEMT	8) SIGFPE	9) SIGKILL	10) SIGBUS
11) SIGSEGV	12) SIGSYS	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGURG	17) SIGSTOP	18) SIGTSTP	19) SIGCONT	20) SIGCHLD
21) SIGTTIN	22) SIGTTOU	23) SIGIO	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGINFO	30) SIGUSR1
31) SIGUSR2				

(obtained using “kill -l”)

The concept of signal

- Signals are mostly used to notify events
  - From the OS
    - Timer expiration (SIGALRM), Child termination (SIGCHLD)
    - Program interrupt — Ctrl-C (SIGINT), Broken pipe (SIGPIPE)
    - Terminal disconnection (SIGHUP)
  - From applications
  - From the CPU (exceptions)

The concept of signal

- Signals are mostly used to notify events
  - From the OS
    - Timer expiration (SIGALRM), Child termination (SIGCHLD)
    - Program interrupt — Ctrl-C (SIGINT), Broken pipe (SIGPIPE)
    - Terminal disconnection (SIGHUP)
  - From applications
    - “please terminate” (SIGTERM)
    - “die!” (SIGKILL)
    - SIGUSR1, SIGUSR2
    - Job control (SIGSTOP, SIGCONT)
  - From the CPU (exceptions)

The concept of signal

- Signals are mostly used to notify events
  - From the OS
    - Timer expiration (SIGALRM), Child termination (SIGCHLD)
    - Program interrupt — Ctrl-C (SIGINT), Broken pipe (SIGPIPE)
    - Terminal disconnection (SIGHUP)
  - From applications
    - “please terminate” (SIGTERM)
    - “die!” (SIGKILL)
    - SIGUSR1, SIGUSR2
    - Job control (SIGSTOP, SIGCONT)
  - From the CPU (exceptions)
    - Segmentation fault (SIGSEGV)
    - Illegal Instruction (SIGILL)
    - Dereferencing misaligned pointer (SIGBUS)
    - Arithmetic error (SIGFPE)

The concept of signal

- Signals are asynchronous
  - No guarantee of express delivery (except for CPU exceptions)
- On the destination process, a signal can be either
  - Delivered
  - Ignored
  - Blocked (i.e. postponed)
- Not every signal can be ignored or postponed
  - SIGKILL, SIGSTOP

The concept of signal

- Each signal has its own *default behavior*
  - Many signals lead to process termination
    - SIGTERM, SIGINT, SIGALRM, … and of course SIGKILL
  - Some lead to termination + core dump file generation
    - SIGSEGV, SIGFPE, SIGBUS, SIGQUIT, SIGABRT, …
  - Some others are ignored
    - SIGCHLD

Sending signals explicitly

- Shell command
  - `kill -SIG pid`
    - `kill -INT 62354`
    - `kill -9 37463 / kill -KILL 37463`

Sending signals explicitly

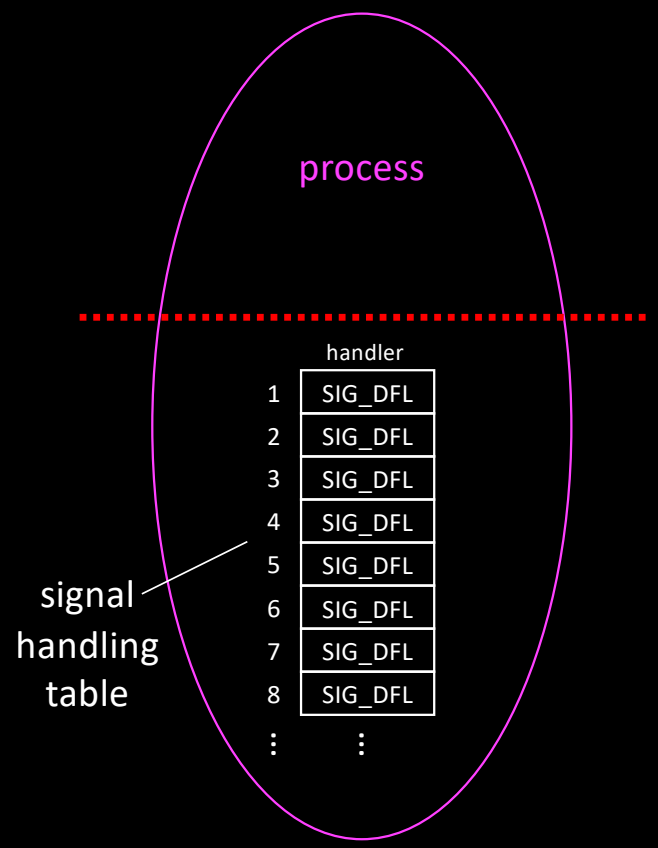
- Shell command
  - `kill -SIG pid`
    - `kill -INT 62354`
    - `kill -9 37463 / kill -KILL 37463`
- System calls
  - `int raise(int sig);`
    - Send signal to ourself (current process)
  - `int kill(pid_t pid, int sig);`
    - Send signal <sig> to process <pid>

Sending signals explicitly

- sig.c

Signal handling

- The kernel maintains a *signal handling table* per process
  - The handler field specifies how to deliver each signal
    - SIG_DFL: default behavior
      - Signal specific
    - SIG_IGN: ignore
    - void (*func)(int): user-level handler
- Changing the handler associated to a given signal: `sigaction` syscall

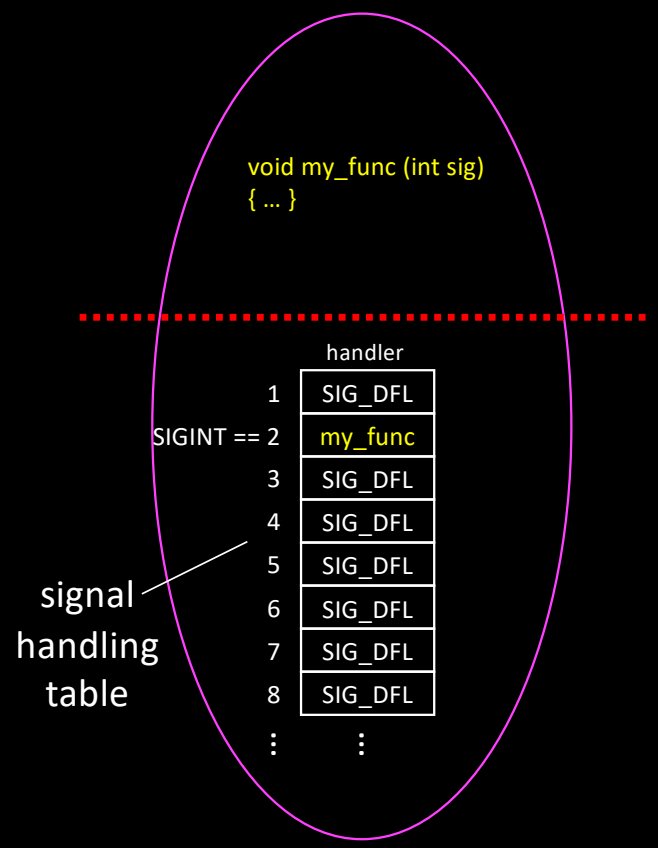


Signal handling

- Changing the handler associated to a given signal:
  - Fill a structure and call `sigaction`

```
struct sigaction sa, old;

sa.sa_flags = 0;
sigemptyset(&sa.sa_mask);
sa.sa_handler = my_func;

sigaction (SIGINT, &sa, &old);
```

To be discussed later



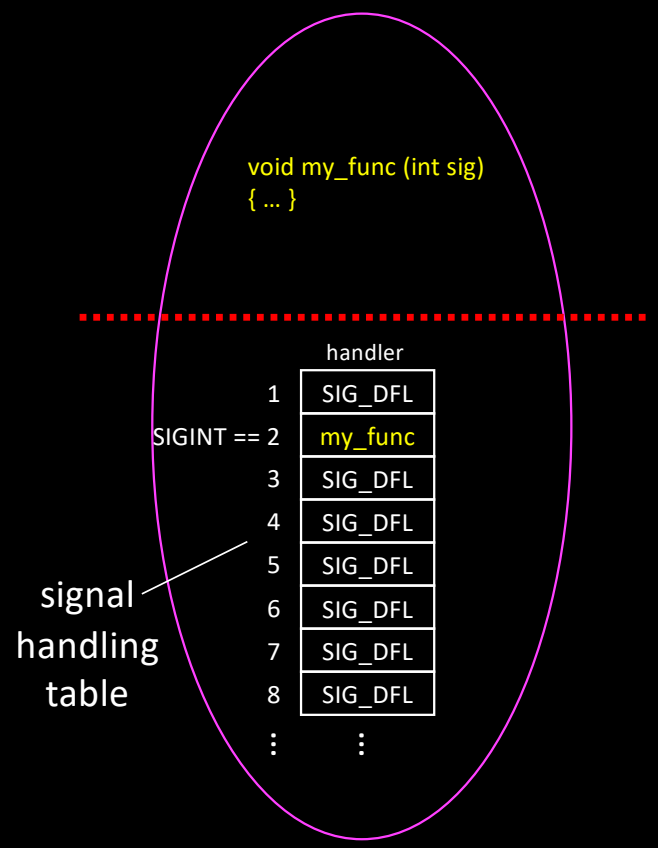
Signal handling

- Changing the handler associated to a given signal:
  - Sometimes, I will use a simplified syntax:

```
sigaction (SIGINT, my_func);
```

- When the signal is delivered
  - The handler is called as if the function was inserted between previous and current instruction

- catch.c, slow-catch.c



Signal Handling

- Temporary catch of signals
  - Catch (SIGINT)
  - Sleep (5)
  - Uncatch (SIGINT)
  - Sleep (5)

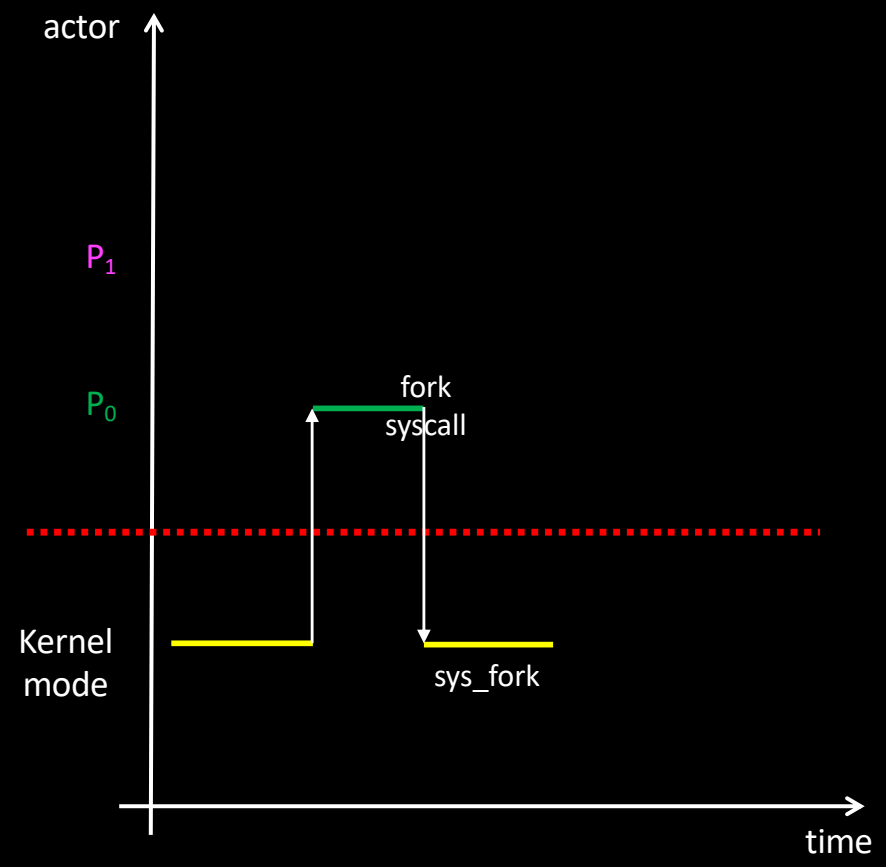
- tempo-catch.c

Catching SIGCHLD

- SIGCHLD is sent to parent each time a process terminates
  - But SIGCHLD is ignored by default
- A process can setup a handler to catch SIGCHLD
  - And call wait() to eliminate zombies
- child.c

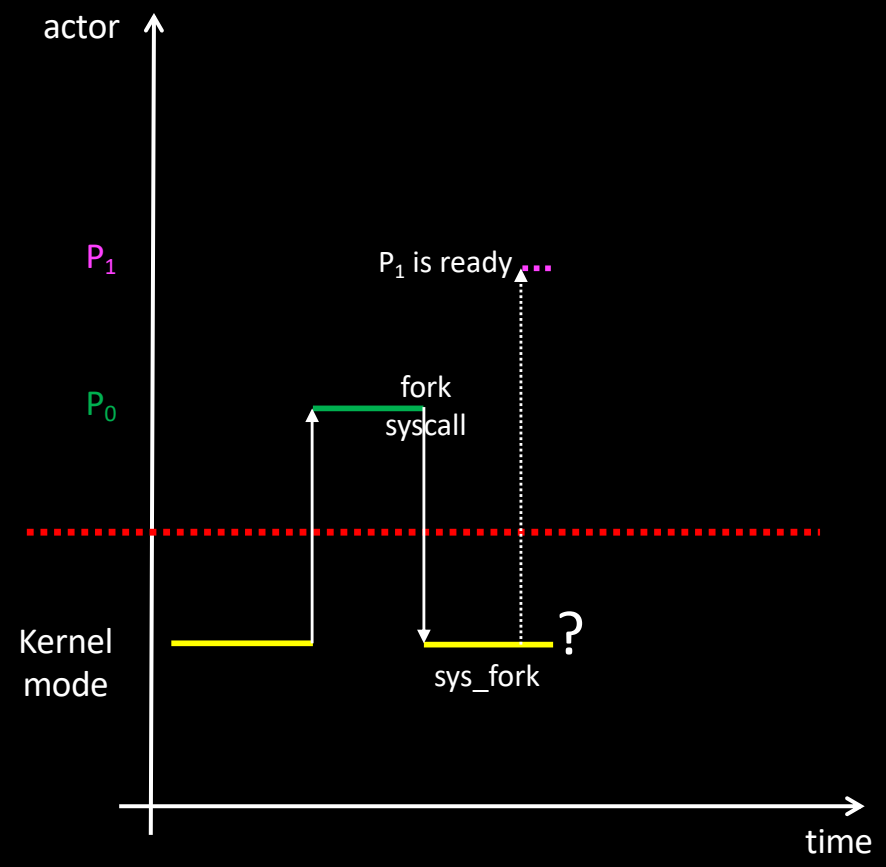
Catching SIGCHLD

- A child can terminate quickly…
  …very quickly



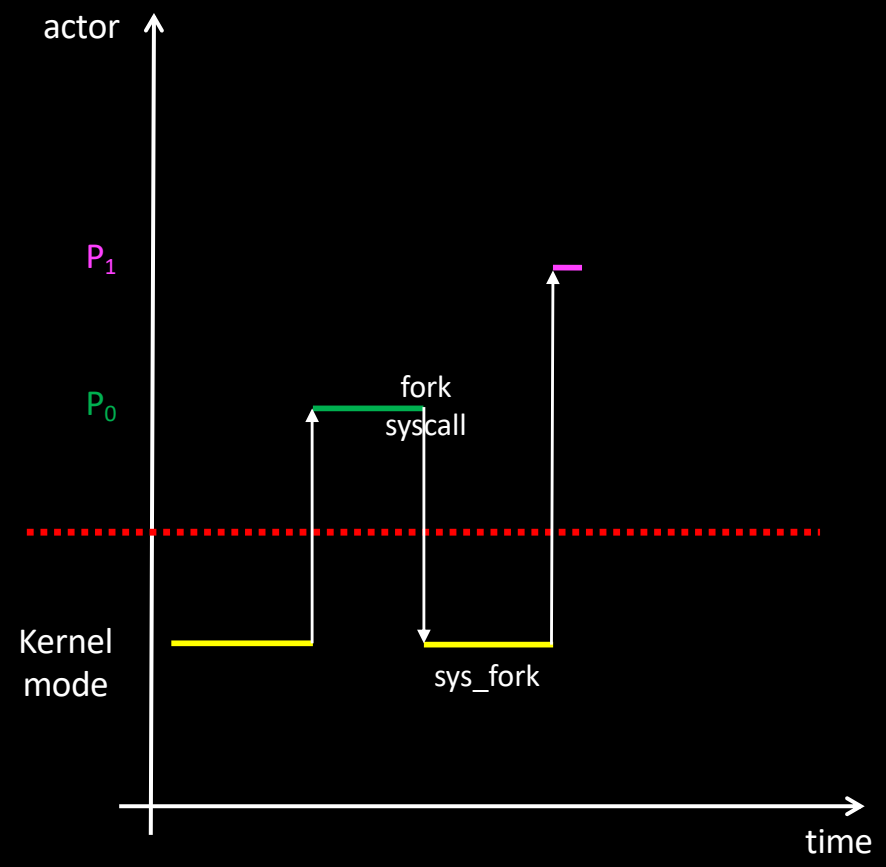
Catching SIGCHLD

- A child can terminate quickly…
  …very quickly



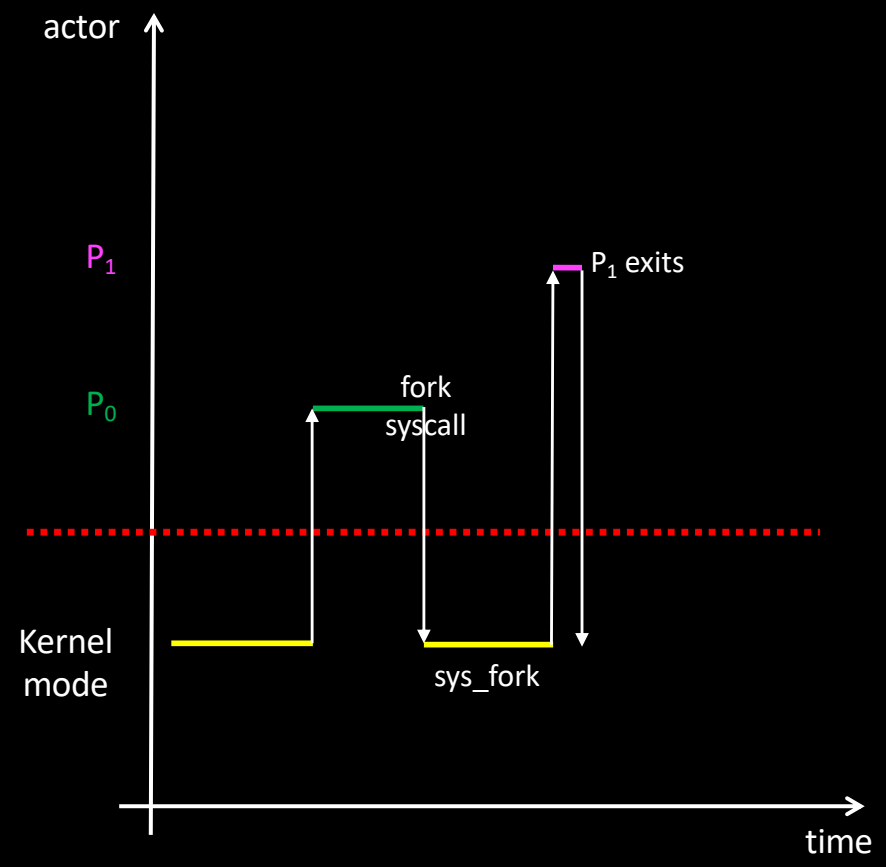
Catching SIGCHLD

- A child can terminate quickly…
  …very quickly



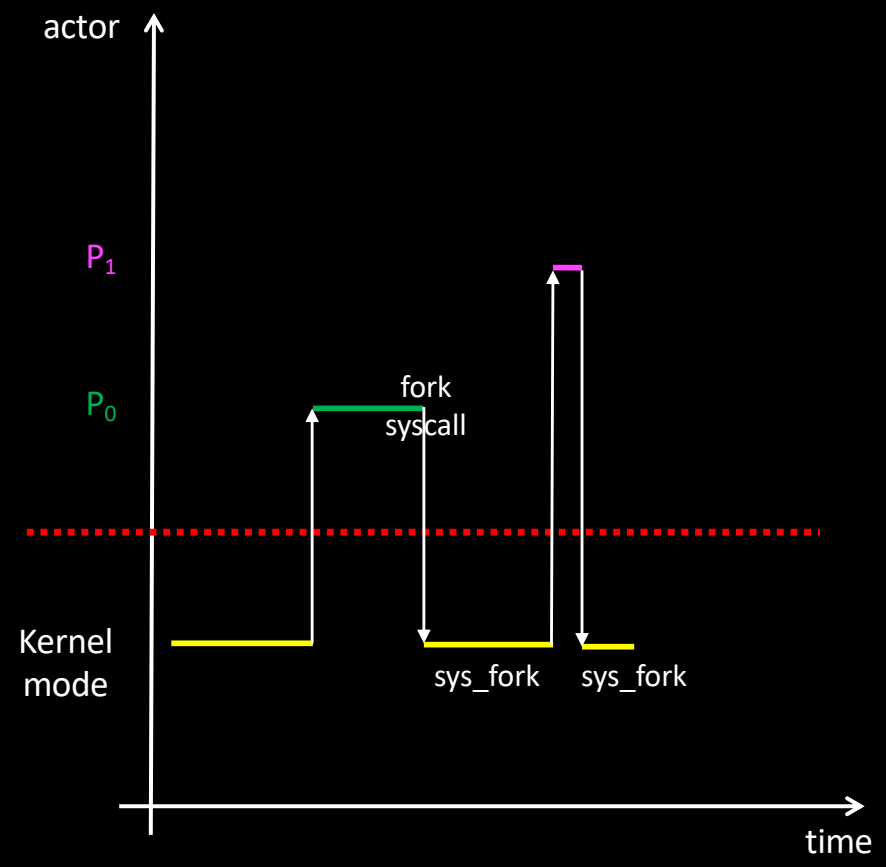
Catching SIGCHLD

- A child can terminate quickly…
  …very quickly



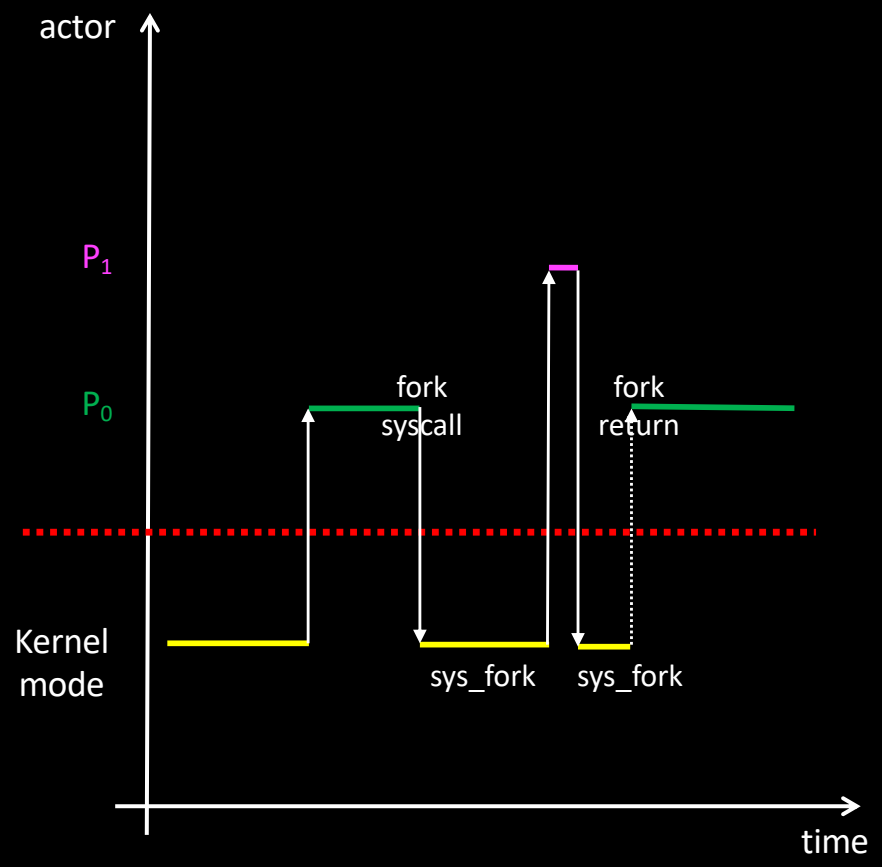
Catching SIGCHLD

- A child can terminate quickly…
  …very quickly



Catching SIGCHLD

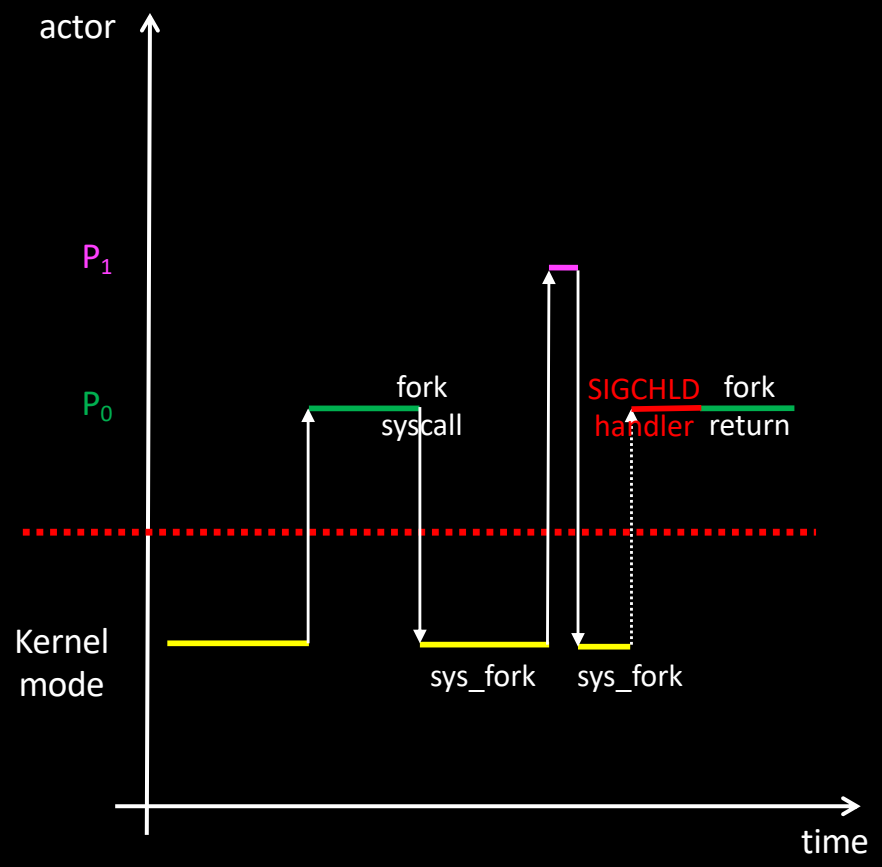
- A child can terminate quickly…
  …very quickly



Catching SIGCHLD

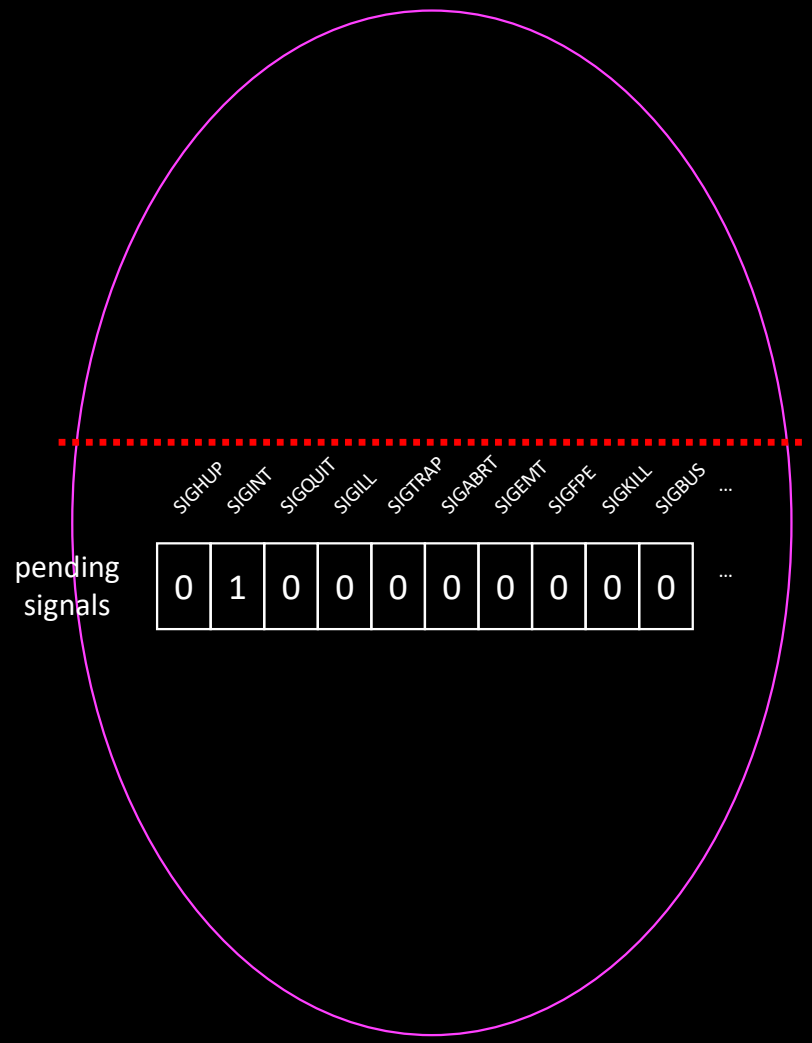
- A child can terminate quickly…
  …very quickly

- So SIGCHLD can be delivered very soon…
  - Back to child.c ?



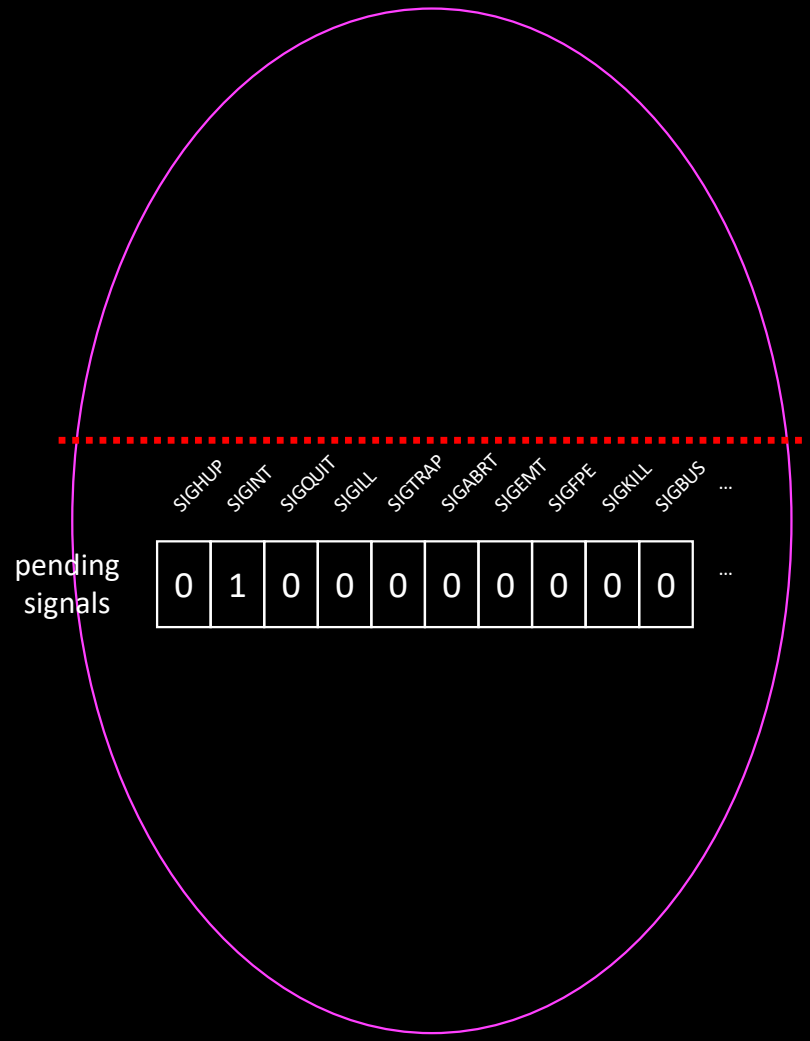
Pending signals

- Signal delivery is asynchronous
  - The kernel simply sets a “pending” bit in a table!
  - Signal is delivered “when possible”



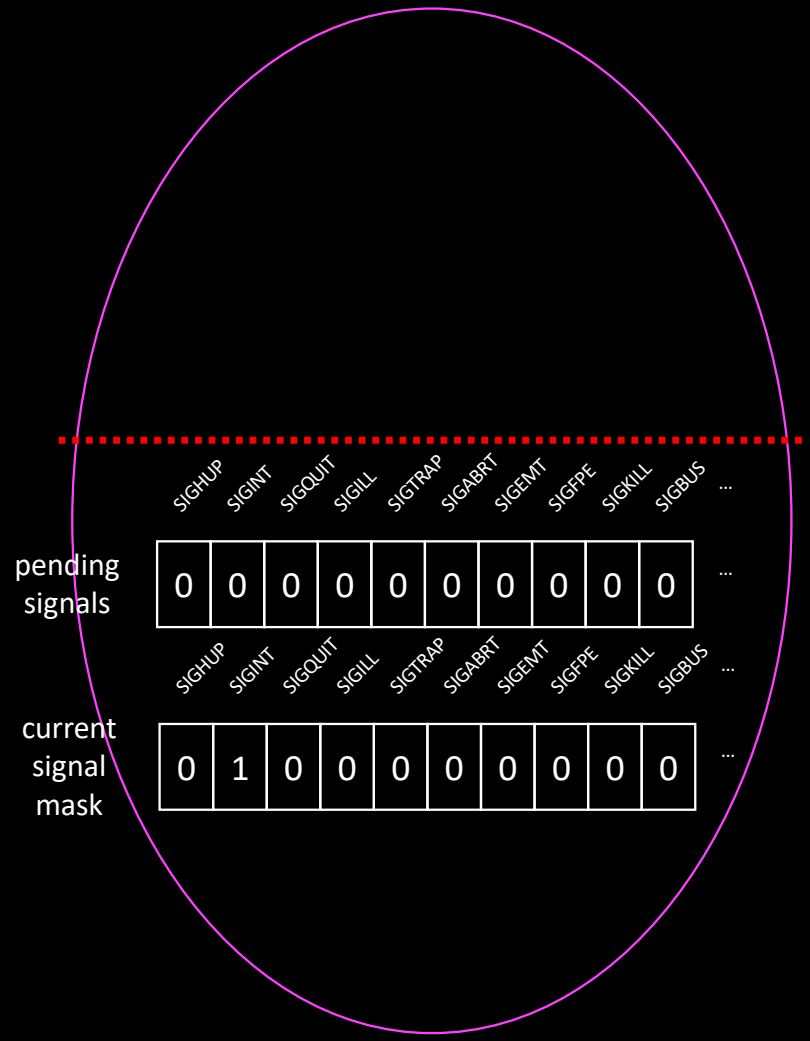
Pending signals

- Signal delivery is asynchronous
  - The kernel simply sets a “pending” bit in a table!
  - Signal is delivered “when possible”
- As a consequence
  - If two signals are sent in a very close time interval…
    - One may be lost!



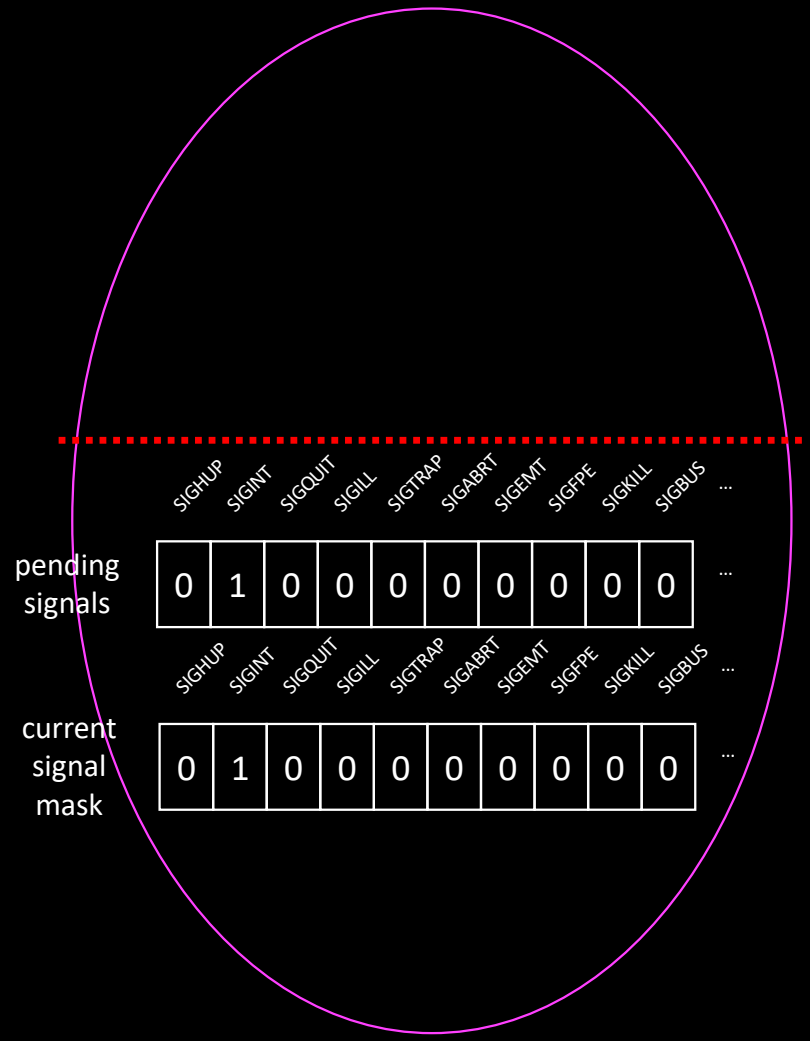
Blocking signals

- Sometimes, the delivery of a signal is not desirable
  - During the update of a complex data structure
  - When the process is not (yet) ready to catch signals
- A process can postpone (i.e. *block*) the delivery of signals
  - The kernel maintains an array of bits
    - Current signal mask



Blocking signals

- Sometimes, the delivery of a signal is not desirable
  - During the update of a complex data structure
  - When the process is not (yet) ready to catch signals
- A process can postpone (i.e. *block*) the delivery of signals
  - The kernel maintains an array of bits
    - Current signal mask
  - Blocked signals will stay “pending” until delivery is allowed again

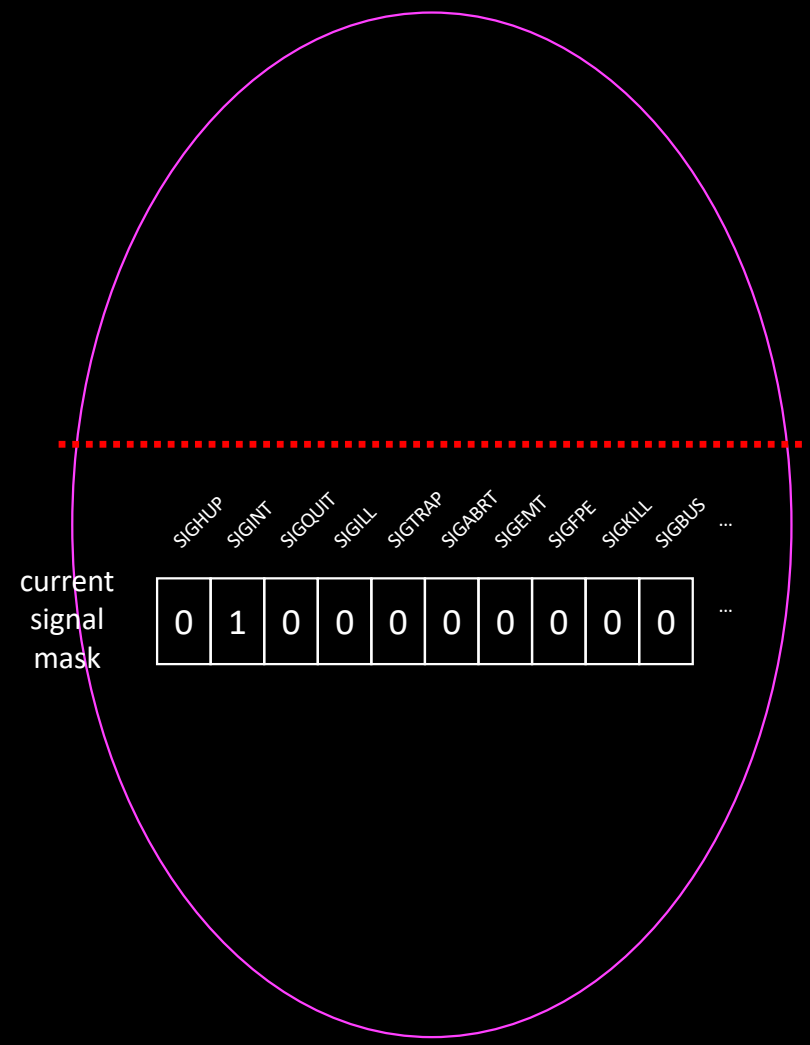


Blocking signals

- The `sigprocmask` syscall modifies the current signal mask

```
int sigprocmask(int how, sigset_t *set,
                         sigset_t *oset);
```

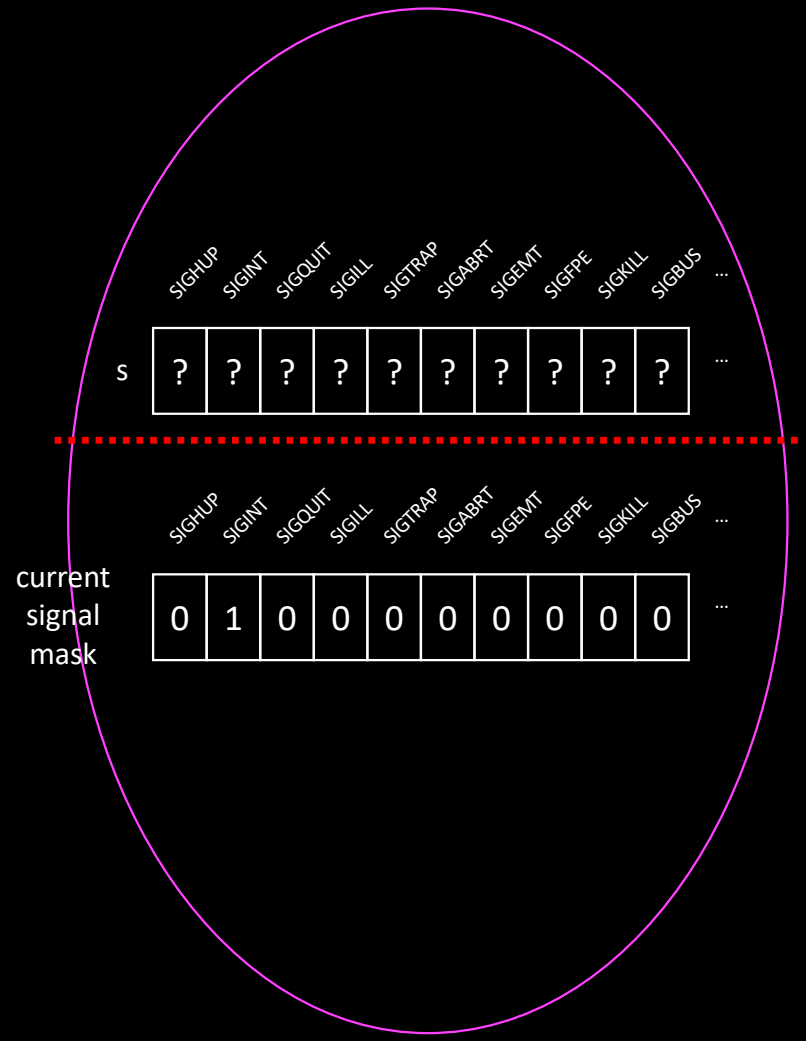
- 2 steps
  - Construct a mask (set) in user space
  - Call sigprocmask and tell how to combine the provided mask with the current signal mask:
    - SIG_SETMASK: replace
    - SIG_BLOCK: block the signals marked with 1, keep others as is
    - SIG_UNBLOCK: unblock signals marked with 1, keep others as is



Blocking signals

- Example: blocking SIGQUIT

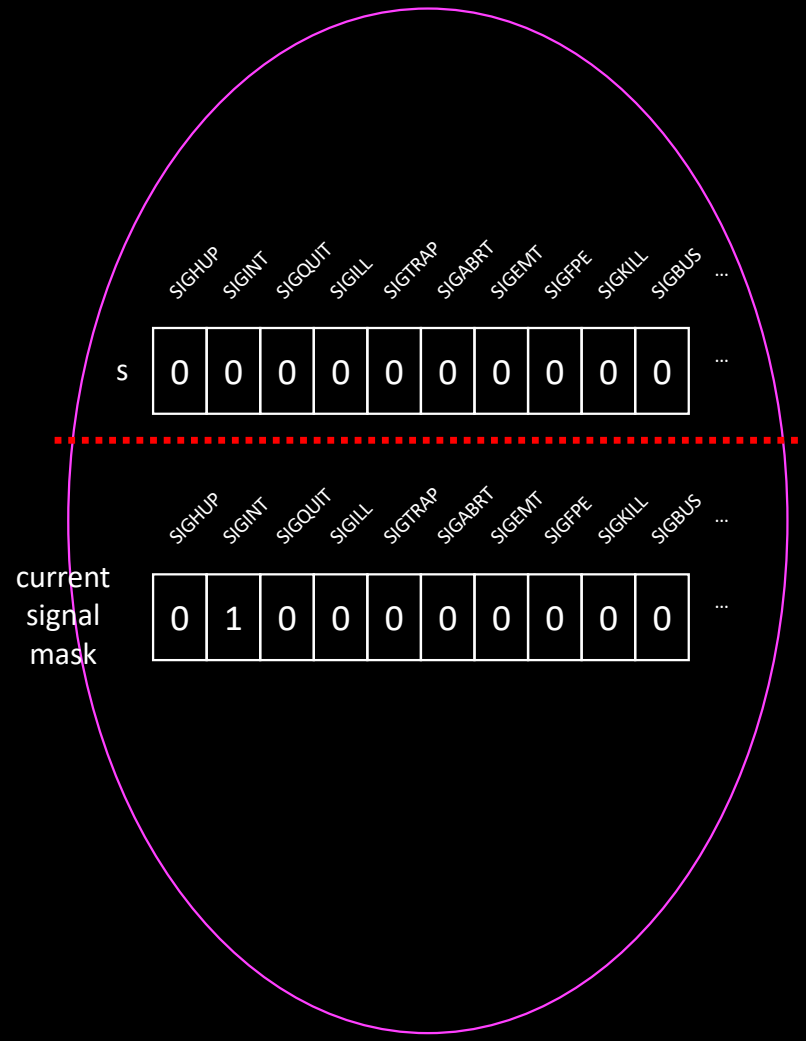
```
sigset_t s;
```



Blocking signals

- Example: blocking SIGQUIT

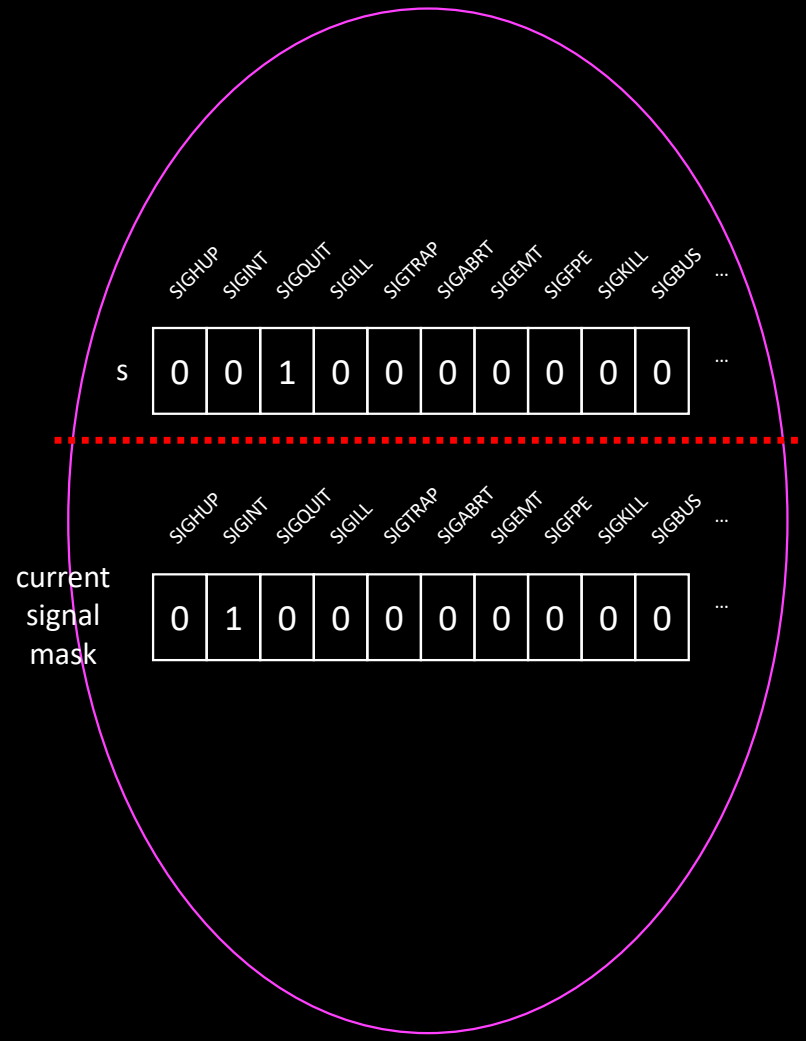
```
sigset_t s;
sigemptyset (&s);
```



Blocking signals

- Example: blocking SIGQUIT

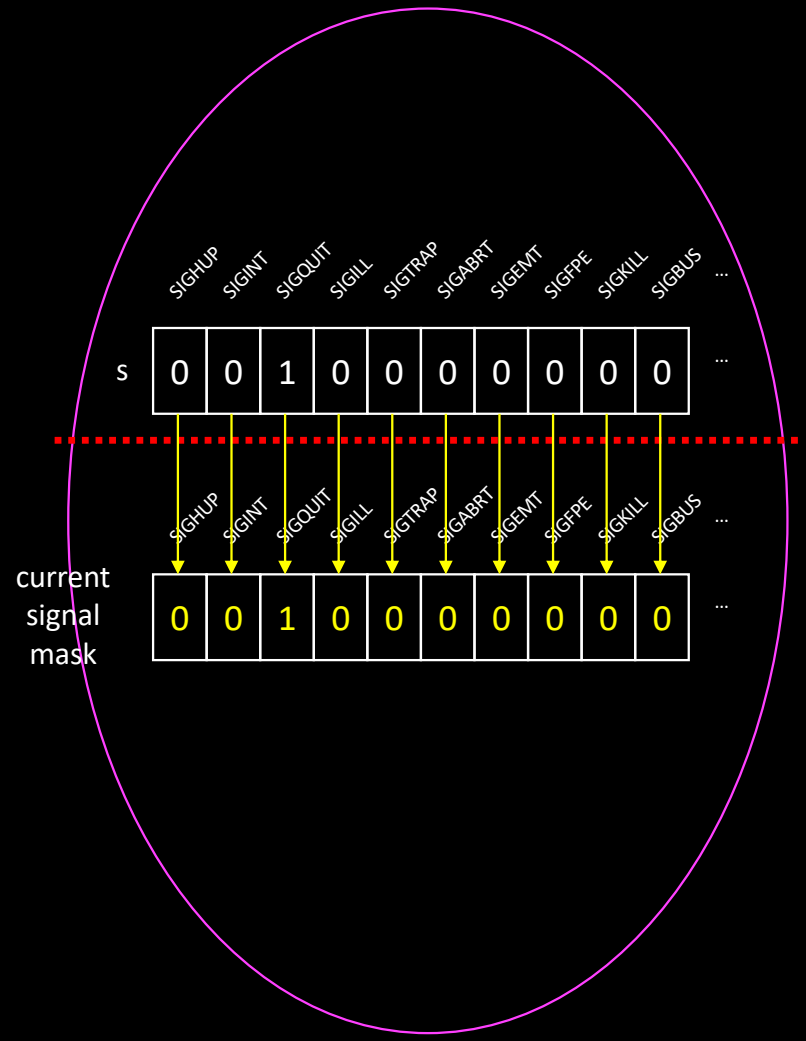
```
sigset_t s;
sigemptyset (&s);
sigaddset (&s, SIGQUIT);
```



Blocking signals

- Example: blocking SIGQUIT

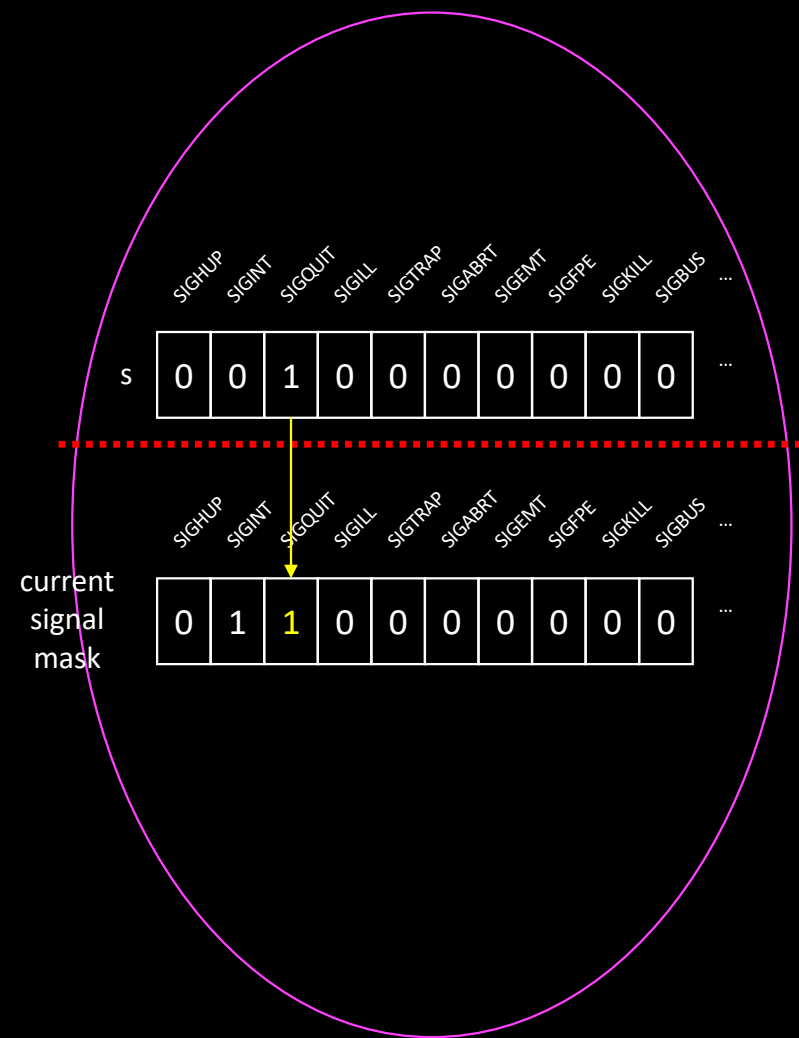
```
sigset_t s;
sigemptyset (&s);
sigaddset (&s, SIGQUIT);

sigprocmask (SIG_SETMASK, &s, NULL);
```



Blocking signals

- Example: blocking SIGQUIT

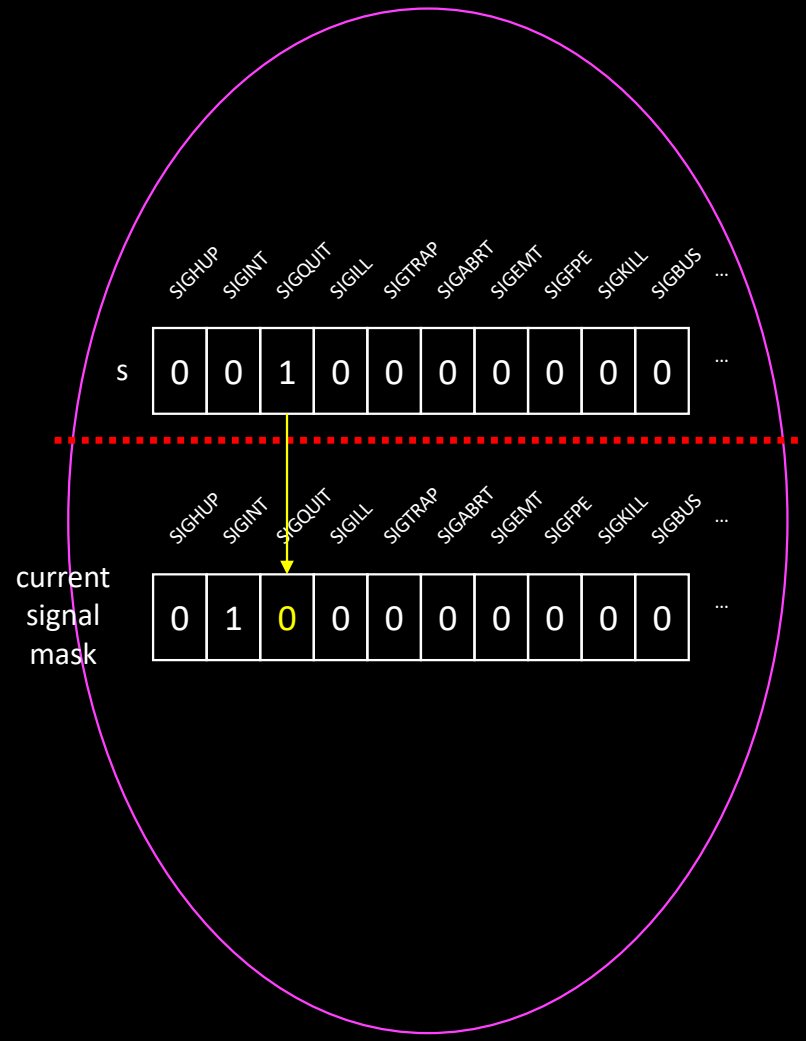
```
sigset_t s;
sigemptyset (&s);
sigaddset (&s, SIGQUIT);

sigprocmask (SIG_BLOCK, &s, NULL);
```



Blocking signals

- Example: unblocking SIGQUIT

```
sigset_t s;
sigemptyset (&s);
sigaddset (&s, SIGQUIT);

sigprocmask (SIG_UNBLOCK, &s, NULL);
```



Blocking signals

- See mask.c

Blocking signals

- During the execution of a signal handler, the corresponding signal is automatically blocked
  - No ”recursive” invocations of handler
  - Additional signals can be blocked
    - Purpose of sigaction struct’s `sa_mask` field

Handling exceptions

- CPU exceptions trap into the kernel
- In turn, the kernel sends a signal to current process
  - SIGSEGV, SIGILL, SIGFPE
- What happens when we install a handler for such signals ?
  - See fault.c

```
int *ptr = NULL;

void my_sig_handler (int sig)
{
  printf ("I received signal %s\n",
          strsignal (sig));
}

int main (int argc, char *argv[])
{

  sigaction (SIGSEGV, my_sig_handler);

  *ptr = 12;

  return 0;
}
```

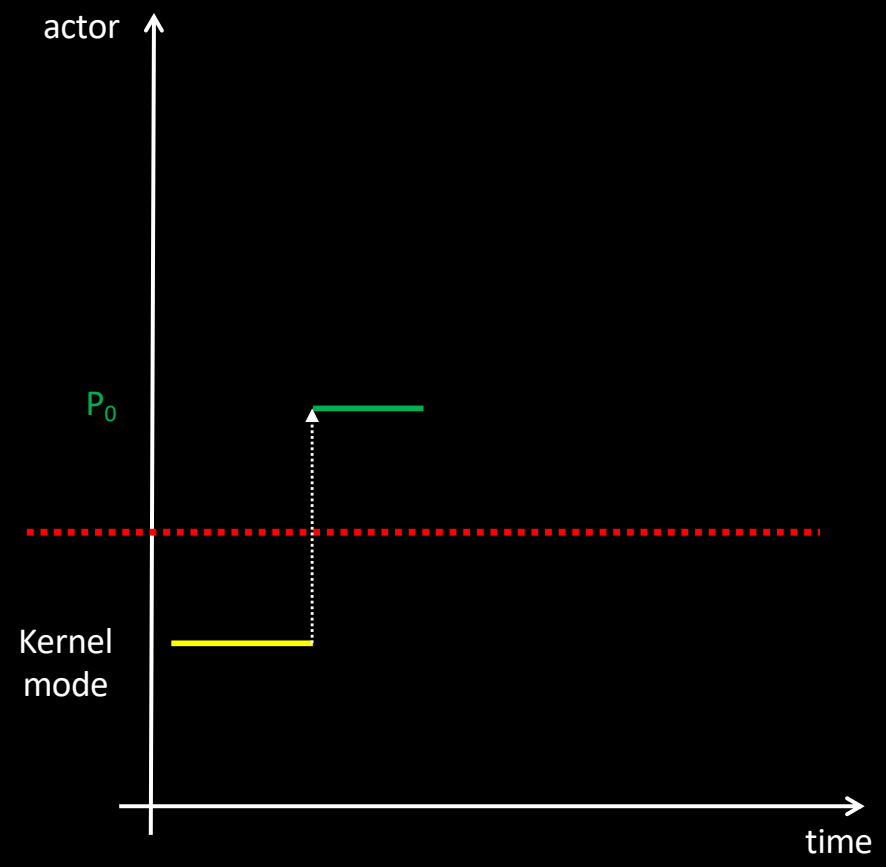
Handling exceptions

- CPU exceptions trap into the kernel
- In turn, the kernel sends a signal to current process
  - SIGSEGV, SIGILL, SIGFPE
- What happens when we install a handler for such signals ?
  - See fault.c
    - Infinite loop…

```
int *ptr = NULL;

void my_sig_handler (int sig)
{
  printf ("I received signal %s\n",
          strsignal (sig));
}

int main (int argc, char *argv[])
{

  sigaction (SIGSEGV, my_sig_handler);

  *ptr = 12;

  return 0;
}
```

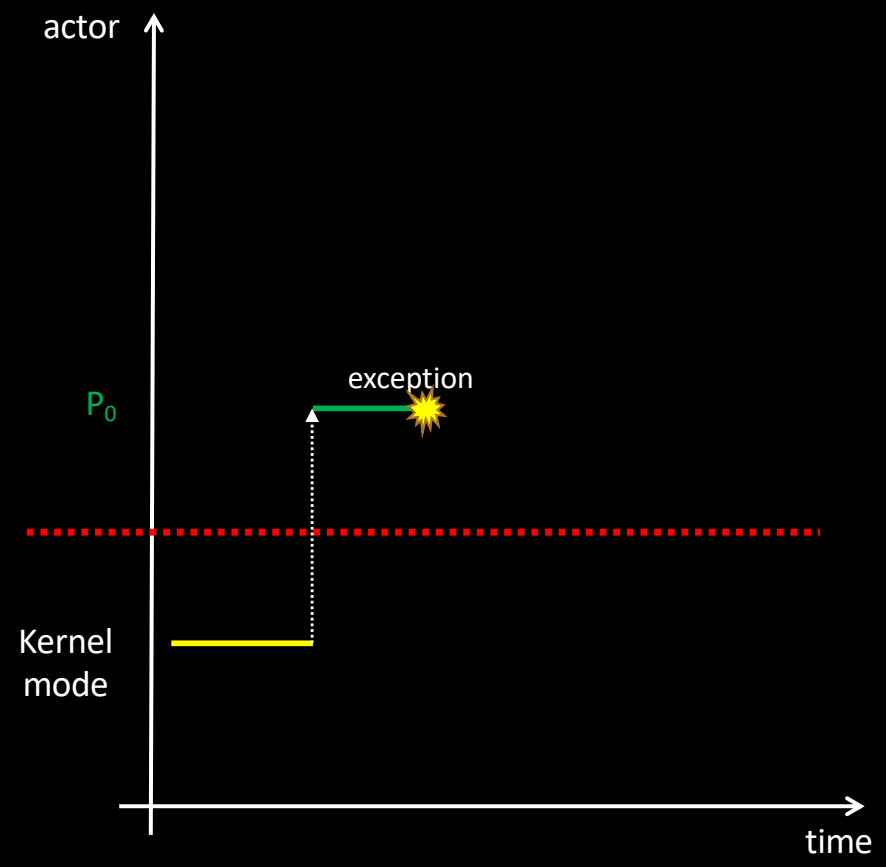
Handling exceptions

- When the CPU attempts to perform a faulty instruction



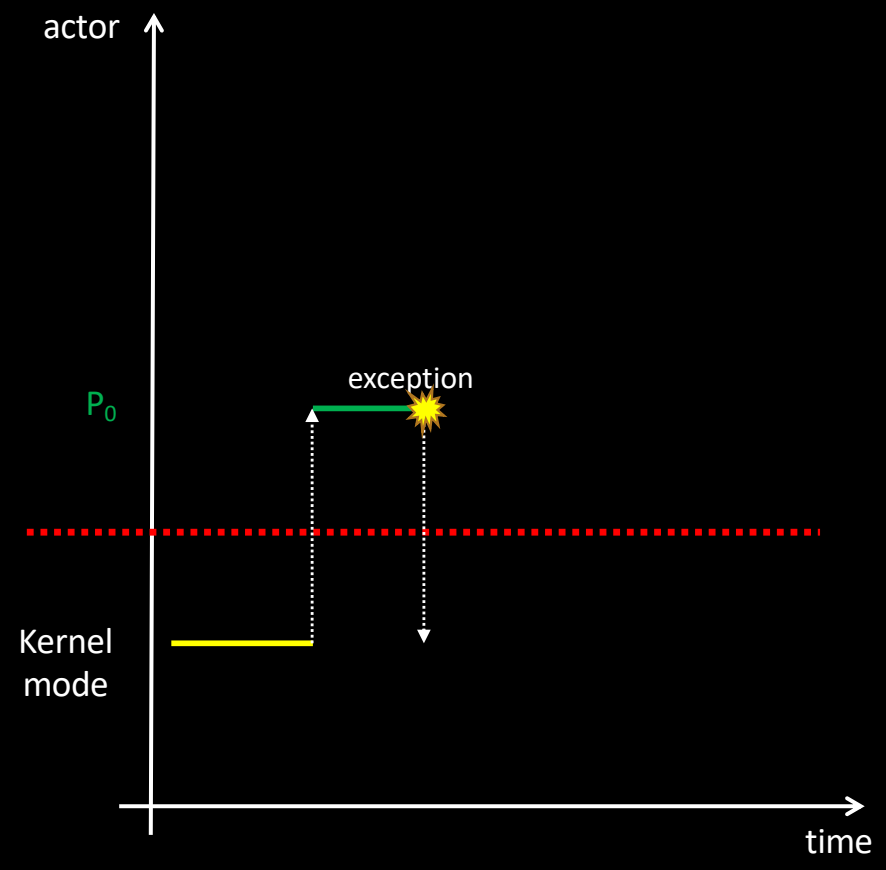
Handling exceptions

- When the CPU attempts to perform a faulty instruction
  - An exception is raised



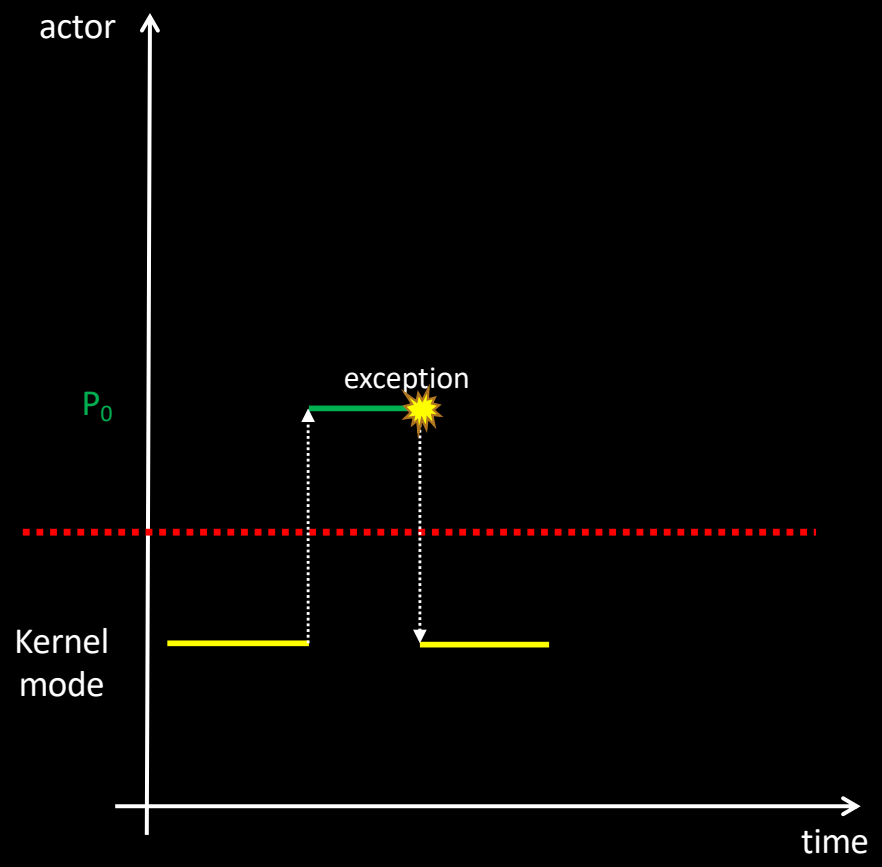
Handling exceptions

- When the CPU attempts to perform a faulty instruction
  - An exception is raised
  - Execution traps into the kernel



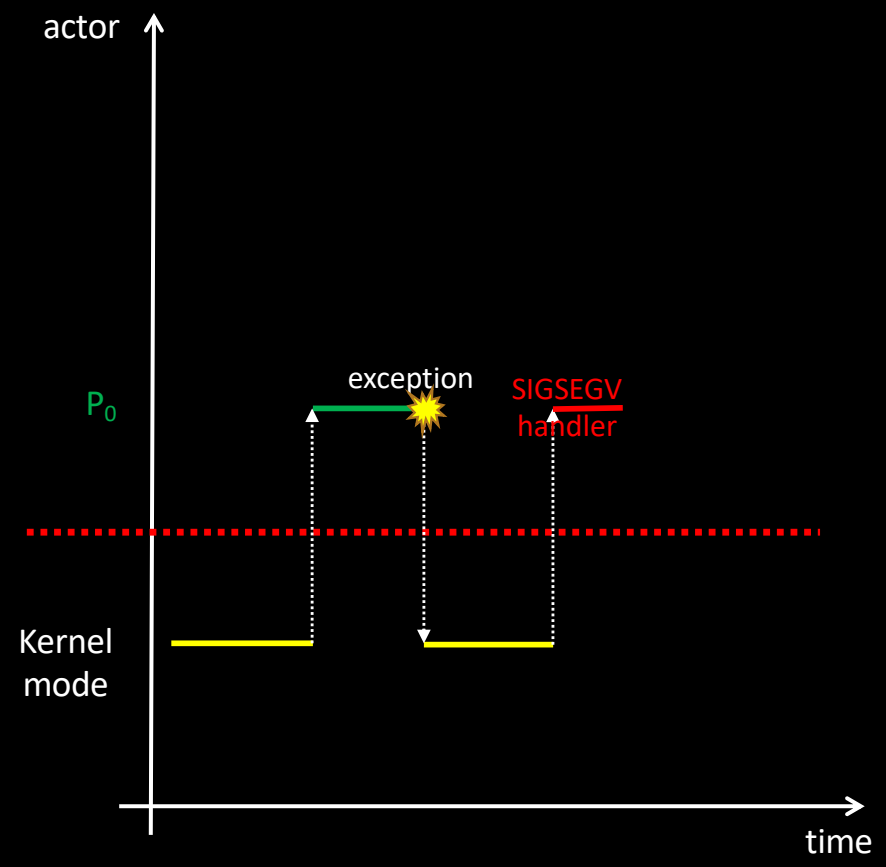
Handling exceptions

- When the CPU attempts to perform a faulty instruction
  - An exception is raised
  - Execution traps into the kernel
  - Kernel checks if a signal handler is installed
    - If so, it modifies the return path so that the handler gets executed when returning to user space



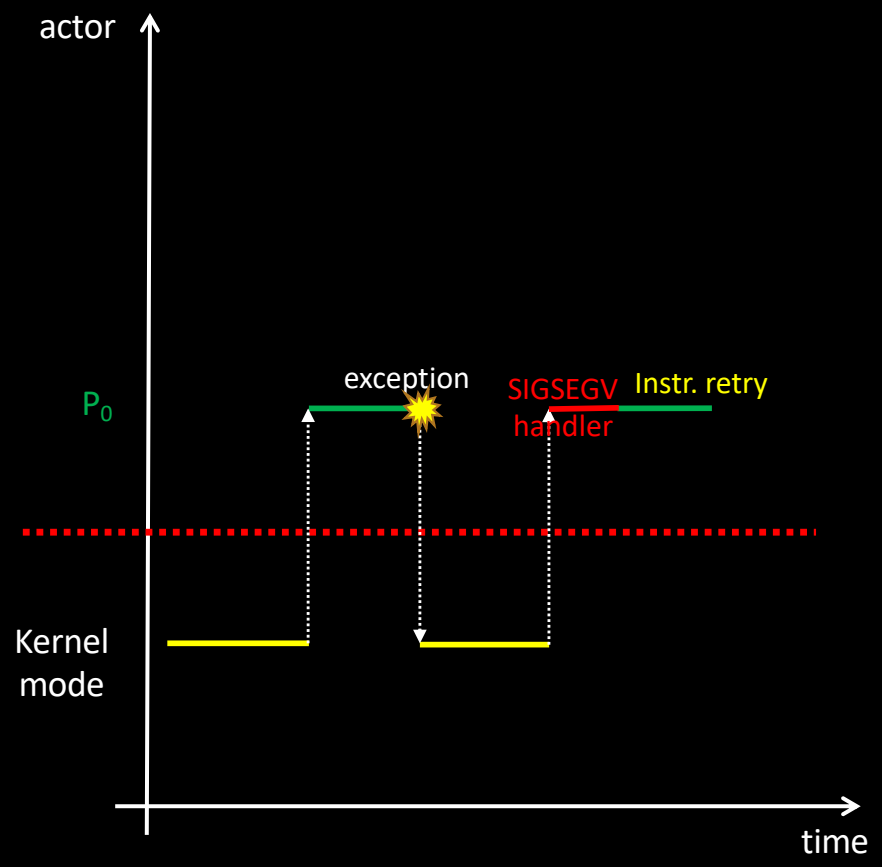
Handling exceptions

- When the CPU attempts to perform a faulty instruction
  - An exception is raised
  - Execution traps into the kernel
  - Kernel checks if a signal handler is installed
    - If so, it modifies the return path so that the handler gets executed when returning to user space
  - Signal handler is executed



Handling exceptions

- When the CPU attempts to perform a faulty instruction
  - An exception is raised
  - Execution traps into the kernel
  - Kernel checks if a signal handler is installed
    - If so, it modifies the return path so that the handler gets executed when returning to user space
  - Signal handler is executed
  - Faulty instruction is retried



Handling exceptions

- Let's try to fix it!
- In the handler, we change the value of the `ptr` variable…
  - See fixfault.c

```
static int *ptr = NULL;
static int glob;

void my_sig_handler (int sig)
{
  pprintf ("ptr was %p\n", ptr);
  ptr = &glob;
  pprintf ("Fixing ptr to %p\n", ptr);
}

int main (int argc, char *argv[])
{
  struct sigaction sa;

  sigaction (SIGSEGV, my_sig_handler);

  *ptr = 12;

  return 0;
}
```

Handling exceptions

- OK, we have a problem!
  - Only the faulty machine instruction gets re-executed
  - Not the C statement!

```
*ptr = 12;
```

```c
static int *ptr = NULL;
static int glob;

void my_sig_handler (int sig)
{
  pprintf ("ptr was %p\n", ptr);
  ptr = &glob;
  pprintf ("Fixing ptr to %p\n", ptr);
}

int main (int argc, char *argv[])
{
  struct sigaction sa;

  sigaction (SIGSEGV, my_sig_handler);

  *ptr = 12;

  return 0;
}
```

Handling exceptions

- OK, we have a problem!
  - Only the faulty machine instruction gets re-executed
  - Not the C statement!

```
*ptr = 12;

        ≡

movq    _ptr(%rip), %rcx
movl    $12, (%rcx)
```

```c
static int *ptr = NULL;
static int glob;

void my_sig_handler (int sig)
{
  pprintf ("ptr was %p\n", ptr);
  ptr = &glob;
  pprintf ("Fixing ptr to %p\n", ptr);
}

int main (int argc, char *argv[])
{
  struct sigaction sa;

  sigaction (SIGSEGV, my_sig_handler);

  *ptr = 12;

  return 0;
}
```

Handling exceptions

- OK, we have a problem!
  - Only the faulty machine instruction gets re-executed
  - Not the C statement!

```
*ptr = 12;

        ≡

movq    _ptr(%rip), %rcx
movl    $12, (%rcx)
```

faulty instruction

```
static int *ptr = NULL;
static int glob;

void my_sig_handler (int sig)
{
  pprintf ("ptr was %p\n", ptr);
  ptr = &glob;
  pprintf ("Fixing ptr to %p\n", ptr);
}

int main (int argc, char *argv[])
{
  struct sigaction sa;

  sigaction (SIGSEGV, my_sig_handler);

  *ptr = 12;

  return 0;
}
```

Handling exceptions

- OK, we have a problem!
  - Only the faulty machine instruction gets re-executed
  - Not the C statement!

```
*ptr = 12;
```

≡

We would like to restart here

```
movq    _ptr(%rip), %rcx
movl    $12, (%rcx)
```

```c
static int *ptr = NULL;
static int glob;

void my_sig_handler (int sig)
{
  pprintf ("ptr was %p\n", ptr);
  ptr = &glob;
  pprintf ("Fixing ptr to %p\n", ptr);
}

int main (int argc, char *argv[])
{
  struct sigaction sa;

  sigaction (SIGSEGV, my_sig_handler);

  *ptr = 12;

  return 0;
}
```

Handling exceptions

- OK, we have a problem!
  - Only the faulty machine instruction gets re-executed
  - Not the C statement!

```
static int *ptr = NULL;
static int glob;

void my_sig_handler (int sig)
{
  pprintf ("ptr was %p\n", ptr);
  ptr = &glob;
  pprintf ("Fixing ptr to %p\n", ptr);
}

int main (int argc, char *argv[])
{
  struct sigaction sa;

  sigaction (SIGSEGV, my_sig_handler);

  *ptr = 12;

  return 0;
}
```

That is:
we would like
to restart here!

Handling exceptions

- OK, we have a problem!
  - Only the faulty machine instruction gets re-executed
  - We need sort of a “super goto”

```
static int *ptr = NULL;
static int glob;

void my_sig_handler (int sig)
{
  ptr = &glob;
  super_goto label;
}

int main (int argc, char *argv[])
{
  struct sigaction sa;

  sigaction (SIGSEGV, my_sig_handler);

label:
  *ptr = 12;

  return 0;
}
```

Handling exceptions

- OK, we have a problem!
  - Only the faulty machine instruction gets re-executed
  - We need sort of a “super goto”
    - But goto only changes the instruction pointer register

```
static int *ptr = NULL;
static int glob;

void my_sig_handler (int sig)
{
  ptr = &glob;
  super_goto label;
}

int main (int argc, char *argv[])
{
  struct sigaction sa;

  sigaction (SIGSEGV, my_sig_handler);

label:
  *ptr = 12;

  return 0;
}
```

Handling exceptions

- OK, we have a problem!
  - Only the faulty machine instruction gets re-executed
  - We need sort of a “super goto”
    - But goto only changes the instruction pointer register
    - In our case, we also need to change the stack pointer
      - As it would happen upon a “normal return” from the handler

```
static int *ptr = NULL;
static int glob;

void my_sig_handler (int sig)
{
  ptr = &glob;
  super_goto label;
}

int main (int argc, char *argv[])
{
  struct sigaction sa;

  sigaction (SIGSEGV, my_sig_handler);

label:
  *ptr = 12;

  return 0;
}
```

Non-local jumps

- Idea
  - Save processor's state at some point
  - Go back to this point later by restoring processor's state

```
typedef struct {
  …  // space to save registers, etc.
} jmp_buf [1];

jmp_buf my_buf;
```

Non-local jumps

- Setjmp

```
jmp_buf my_buf;
int r = setjmp (my_buf);
```

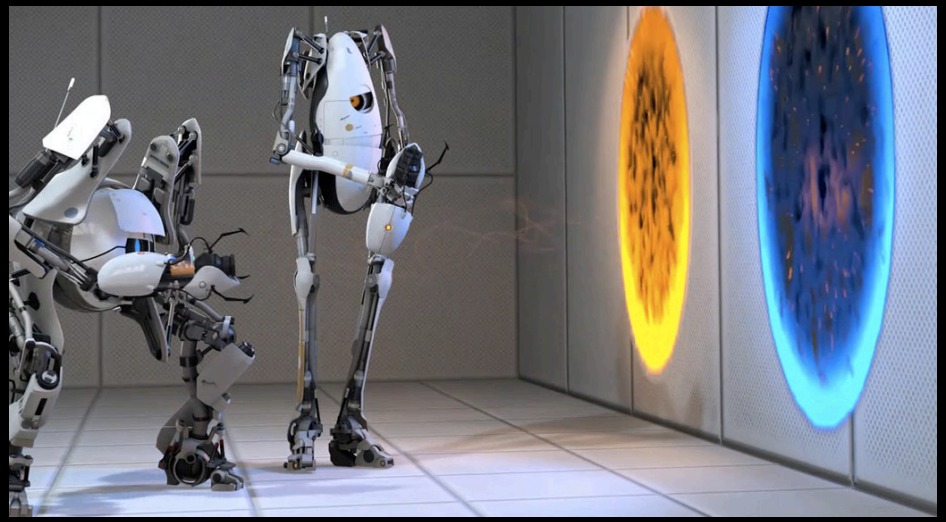
- First call
  - Saves registers into my_buf
  - Returns 0

Non-local jumps

- Setjmp

```
jmp_buf my_buf;
int r = setjmp (my_buf);
```

- First call
  - Saves registers into my_buf
  - Returns 0
- Next returns
  - Come from teleportation!



Non-local jumps

- Setjmp

```
jmp_buf my_buf;
int r = setjmp (my_buf);
```

- First call
  - Saves registers into `my_buf`
  - Returns 0
- Next returns
  - Come from teleportation!

- Longjmp

```
longjmp (my_buf, val);
```

- Restores all registers stored in `my_buf`
  - Thus, control goes back to where instruction pointer was last saved

Non-local jumps

- Setjmp

```
jmp_buf my_buf;
int r = setjmp (my_buf);
```

- First call
  - Saves registers into `my_buf`
  - Returns 0
- Next returns
  - Come from teleportation!

- Longjmp

```
longjmp (my_buf, val);
```

- Restores all registers stored in `my_buf`
  - Thus, control goes back to where instruction pointer was last saved
    - Inside a call to `setjmp`!
    - The return value of `setjmp` is `val`
      - `val` should be ≠ 0

Non-local jumps

- First example
- The following code features an endless loop
  - longjmp “jumps” back to setjmp
- See longjmp.c

```
jmp_buf buf;

int main (int argc, char *argv[])
{
  int r;

  r = setjmp (buf);

  .. // do something

  longjmp (buf, 1);

  return 0;
}
```

Non-local jumps

- We usually use setjmp/longjmp to go back to checkpoints in the code
- Longjmp is used to raise a ”soft exception”
  - Faster than propagating -1 return codes when error detected at a deep nesting level
  - See retry.c

```
r = setjmp (buf);

if (r == 0) {

  compute ();

} else {
  // fix the problem
  …
  // and retry or do something else
  compute ();
}
```

Non-local jumps

- Idea
  - Use longjmp to escape from a signal handler!
- See truefix.c

```
static int *ptr = NULL;
static int glob;

jmp_buf buf;

void my_sig_handler (int sig)
{
  ptr = &glob;
  longjmp (buf, 1);
}
```

Non-local jumps

- Works soooo well…
- Let's use it twice!
  - See fix-and-fix.c

```
sigaction (SIGSEGV, my_sig_handler);

setjmp (buf);
*ptr = 12;

…

ptr = NULL;

…

setjmp (buf);
*ptr = 13;
```

Non-local jumps

- Works soooo well…
- Let's use it twice!
  - See fix-and-fix.c
  - Damned! It doesn't work!
    - The second Segmentation Fault is lethal

```
sigaction (SIGSEGV, my_sig_handler);

setjmp (buf);
*ptr = 12;

…

ptr = NULL;

…

setjmp (buf);
*ptr = 12;
```

Non-local jumps

- Works soooo well…
- Let's use it twice!
  - See fix-and-fix.c
  - Damned! It doesn't work!
    - The second Segmentation Fault is lethal
  - Reason: SIGSEGV has been masked since the first call to the handler

```
sigaction (SIGSEGV, my_sig_handler);

setjmp (buf);
*ptr = 12;

…

ptr = NULL;

…

setjmp (buf);
*ptr = 12;
```

Non-local jumps

- When mixing setjmp/longjmp and signals
  - In addition to CPU registers…
    …the current signal mask should probably be saved/restored as well

- sigsetjmp/siglongjmp

```
sigjmp_buf my_buf;
int r = sigsetjmp (my_buf, 1);
```

  - 0 = regular setjmp
  - 1 = save current signal mask

- See ultimate-fix.c

Non-local jumps

- Caveats
  - Setjmp saves “some CPU registers”
    - What does it mean with respect to variables?
    - Does setjmp capture i, j and k? Maybe one of them?

```
{
  int i, j, k;
  setjmp (buf);
  …
}
```

    - See registers.c

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