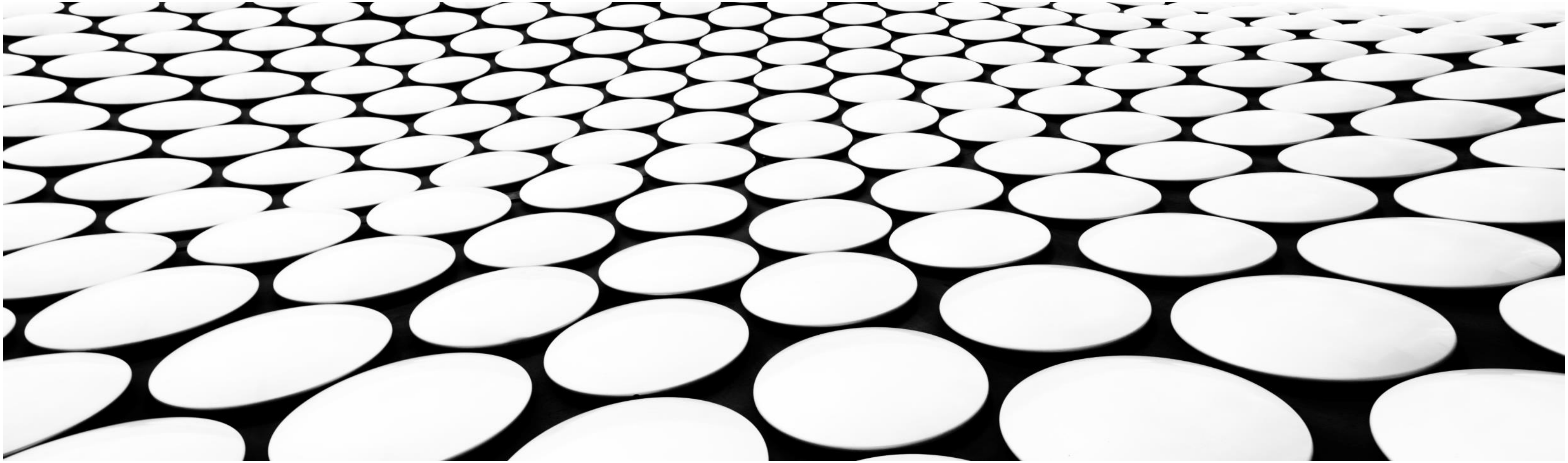
EXCEPTIONS



CONTROL FLOW

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU's *control flow* (or *flow of control*)

Physical control flow

Time

<startup>
$inst_1$
$inst_2$
$inst_3$
…
$inst_n$
<shutdown>

ALTERING THE CONTROL FLOW

- Traditionally, two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return

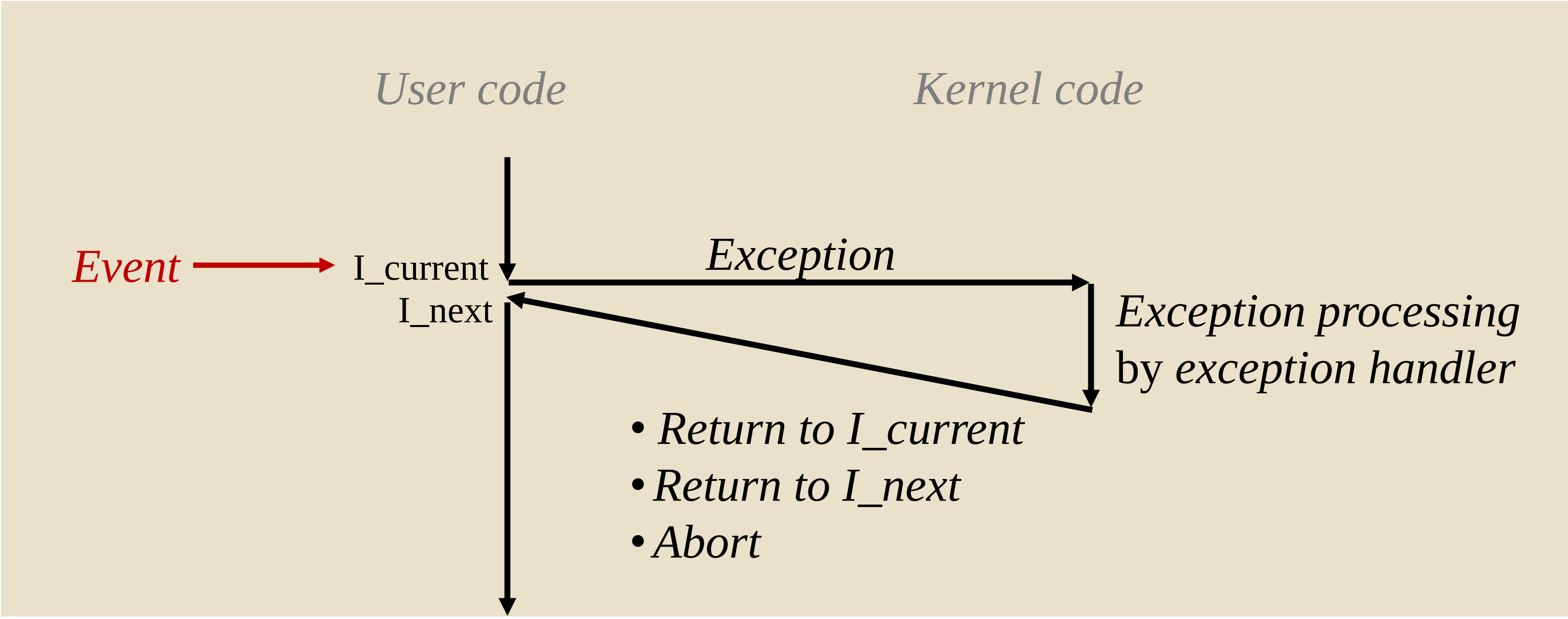
  React to changes in ***program state***
- Insufficient for a useful system:
  Difficult to react to changes in *system state*
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires
- System needs mechanisms for “exceptional control flow”

EXCEPTIONAL CONTROL FLOW

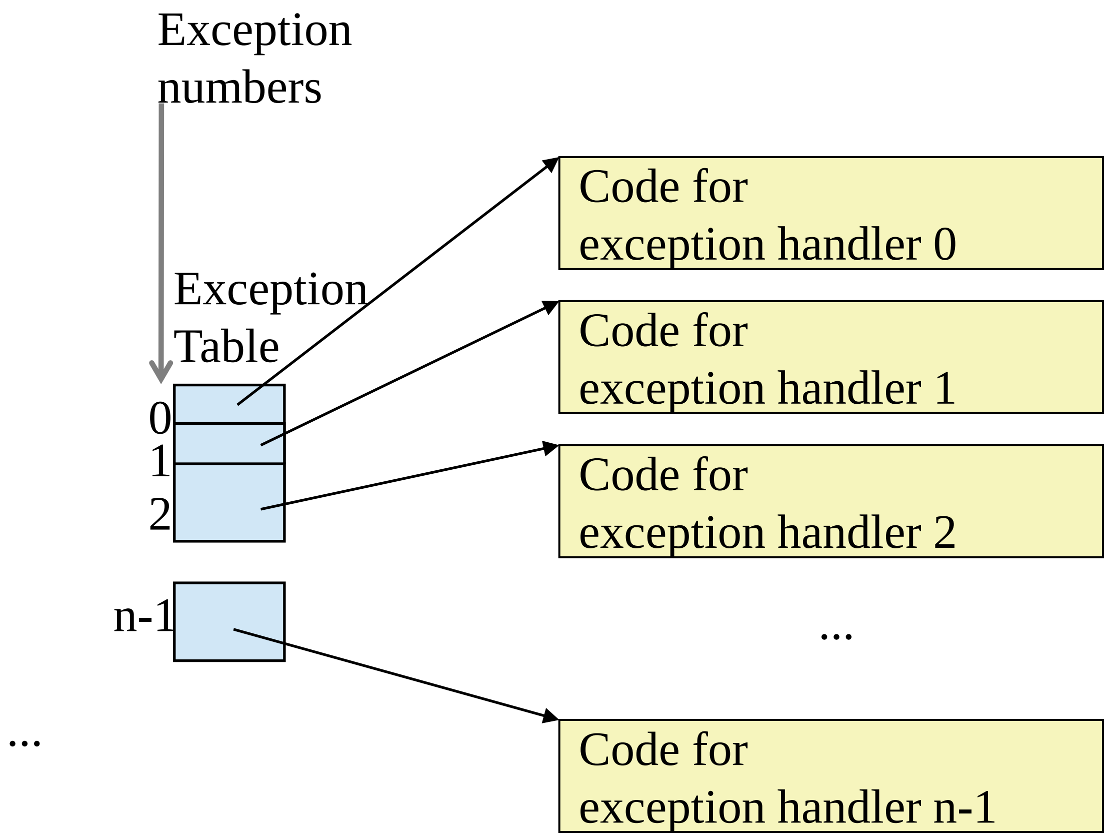
- Exists at all levels of a computer system
- Low level mechanisms
  - 1. **Exceptions**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software
- Higher level mechanisms
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software
  - 4. **Nonlocal jumps**: setjmp() and longjmp()
    - Implemented by C runtime library

EXCEPTIONS

- An *exception* is an abrupt change in the control flow in response to some changes in the processor's state (a transfer of control to the OS *kernel* in response to some *event*)
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



EXCEPTION TABLES



- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

ASYNCHRONOUS EXCEPTIONS (INTERRUPTS)

- Caused by events external to the processor –Network adapters, disk controllers, timer chips
  - Indicated by setting the processor's *interrupt pin*
  - Handler returns to "next" instruction

- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

SYNCHRONOUS EXCEPTIONS

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - Intentional- user program request services
    - Examples: ***system calls***, breakpoint traps, special instructions- fork(), read(), execve()
    - Returns control to "next" instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

EXCEPTIONS IN LINUX/X86-64 SYSTEMS

- Linux/x86-64: Faults and Aborts
  - Divide Error (exception 0): An application attempt to divide by zero or the result of a divide instruction is too big for the destination operand- floating exception
  - General Protection Fault (exception 13): A program references an undefined area of virtual memory or the program attempt to write to a read-only text segment- segmentation Fault
  - Page Fault (exception 14)
  - Machine Check (exception 18): Fatal hardware error

SYSTEM CALLS

- **Each x86-64 system call has a unique ID number correspond to an offset in a jump table in the kernel**
- **Examples:**

Number	*Name*	*Description*
0	read	Read file
1	write	Write file
2	open	Open file
3	close	Close file
4	stat	Get info about file
57	fork	Create process
59	execve	Execute a program
60	_exit	Terminate process
62	kill	Send signal to process

EXCEPTIONS IN LINUX/X86-64 SYSTEMS

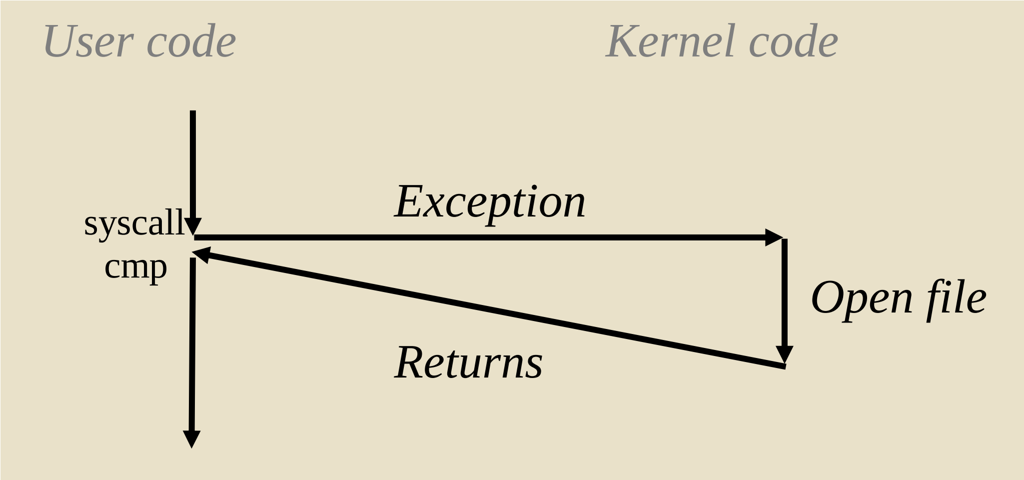
- Linux/x86-64: System call

Number	Name	Description	Number	Name	Description
0	`read`	Read file	33	`pause`	Suspend process until signal arrives
1	`write`	Write file	37	`alarm`	Schedule delivery of alarm signal
2	`open`	Open file	39	`getpid`	Get process ID
3	`close`	Close file	57	`fork`	Create process
4	`stat`	Get info about file	59	`execve`	Execute a program
9	`mmap`	Map memory page to file	60	`_exit`	Terminate process
12	`brk`	Reset the top of the heap	61	`wait4`	Wait for a process to terminate
32	`dup2`	Copy file descriptor	62	`kill`	Send signal to a process

SYSTEM CALL EXAMPLE: OPENING FILE

- User calls: open(filename, options)
- Calls __open function, which invokes system call instruction syscall

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax  # open is syscall #2
e5d7e:   0f 05               syscall         # Return value in %rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```



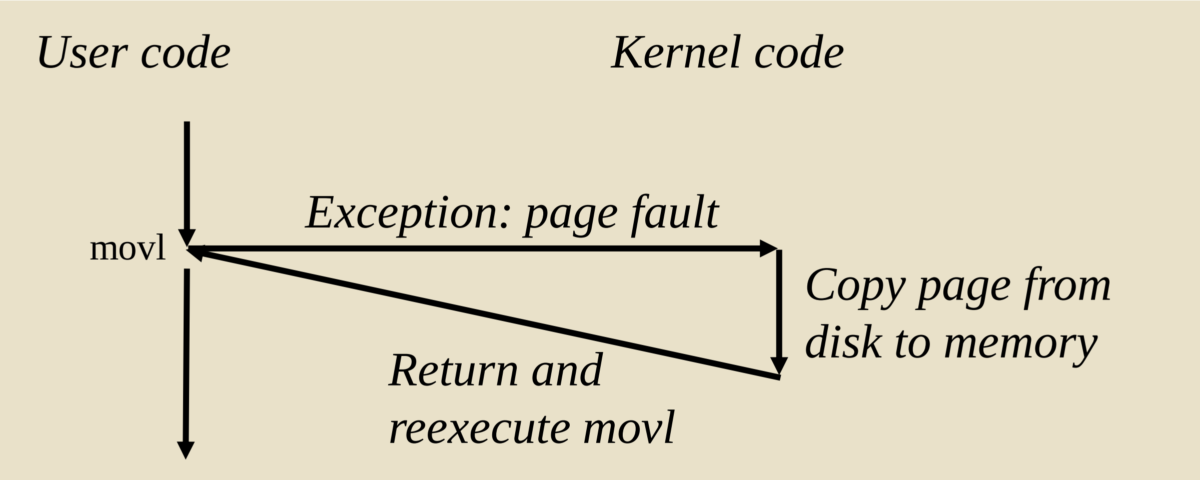
- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

FAULT EXAMPLE: PAGE FAULT

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

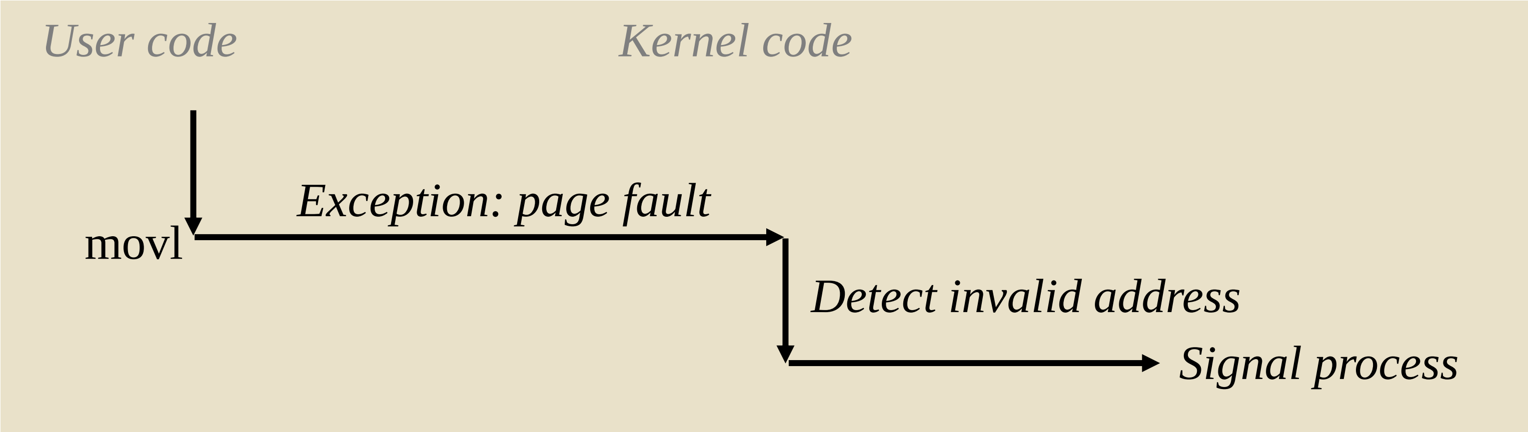
```
80483b7:        c7 05 10 9d 04 08 0d    movl    $0xd,0x8049d10
```



FAULT EXAMPLE: INVALID MEMORY REFERENCE

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7:        c7 05 60 e3 04 08 0d    movl    $0xd,0x804e360
```



- Sends `SIGSEGV` signal to user process
- User process exits with "**segmentation fault**"

SIGNALS

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system
- Low level hardware exceptions are processed by the kernels exception handlers and not visible to user process
- Signals provide a mechanism for exposing the occurrence of such exception to user processes
  - Akin to exceptions and interrupts
  - Sent from the kernel (sometimes at the request of another process) to a process
  - Signal type is identified by small integer ID's (1-30)
  - Only information in a signal is its ID and the fact that it arrived

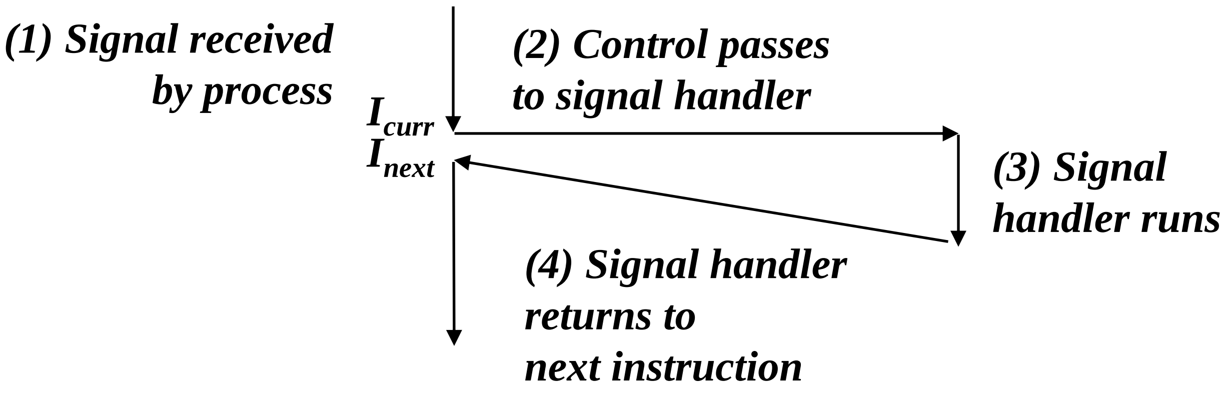
ID	***Name***	***Default Action***	***Corresponding Event***
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

SIGNAL CONCEPTS: SENDING A SIGNAL

- Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process
- Kernel sends a signal for one of the following reasons:
  - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
  - Another process has invoked the **kill** system call to explicitly request the kernel to send a signal to the destination process

SIGNAL CONCEPTS: RECEIVING A SIGNAL

- A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal
- Some possible ways to react:
  - ***Ignore*** the signal (do nothing)
  - ***Terminate*** the process (with optional core dump)
  - ***Catch*** the signal by executing a user-level function called ***signal handler***
    - Akin to a hardware exception handler being called in response to an asynchronous interrupt:



SIGNAL CONCEPTS: PENDING AND BLOCKED SIGNALS

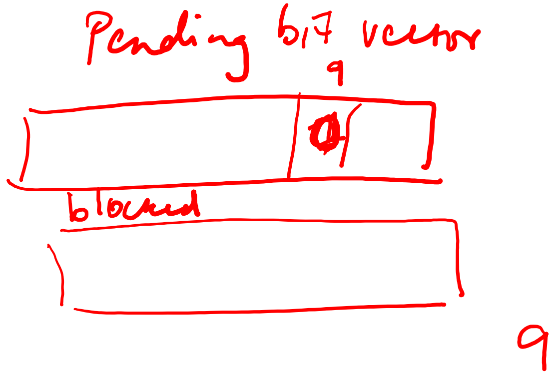
- A signal is *pending* if sent but not yet received
  - There can be at most one pending signal of any particular type
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded

- A process can *block* the receipt of certain signals
  - Blocked signals can be delivered, but will not be received until the signal is unblocked

- A pending signal is received at most once

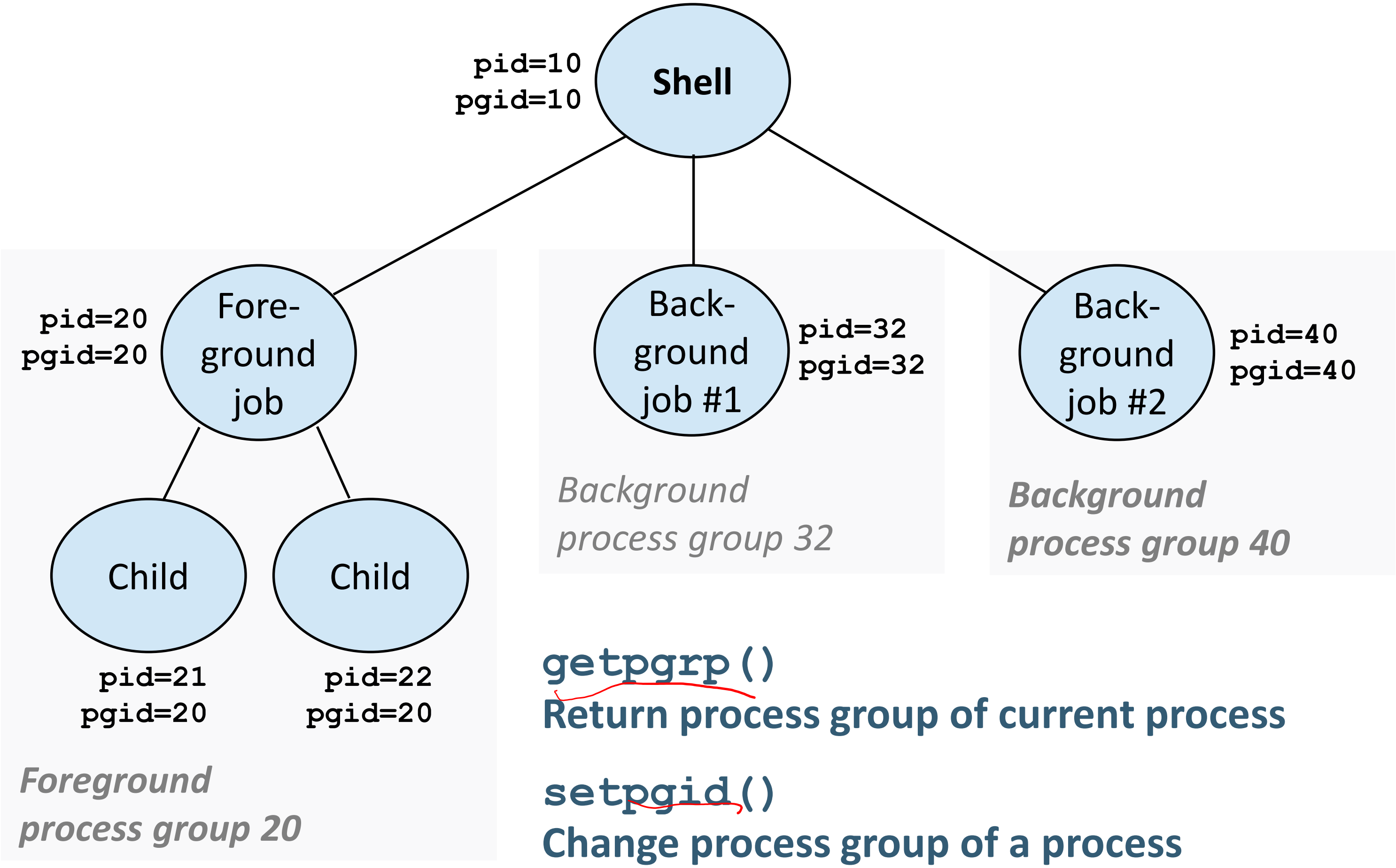
SIGNAL CONCEPTS: PENDING/BLOCKED BITS

- Kernel maintains pending and blocked bit vectors in the context of each process
  - **pending**: represents the set of pending signals
    - Kernel sets bit k in **pending** when a signal of type k is delivered
    - Kernel clears bit k in **pending** when a signal of type k is received
  - **blocked**: represents the set of blocked signals
    - Can be set and cleared by using the **sigprocmask** function
    - Also referred to as the *signal mask*.



SENDING SIGNALS: PROCESS GROUPS

- Every process belongs to exactly one process group



`getpgrp()`
Return process group of current process

`setpgid()`
Change process group of a process

SENDING SIGNALS WITH /BIN/KILL PROGRAM

- /bin/kill program sends arbitrary signal to a process or process group

- Examples
  - **/bin/kill –9 24818**
    Send SIGKILL to process 24818
  - **/bin/kill –9 –24817**
    Send SIGKILL to every process in process group 24817

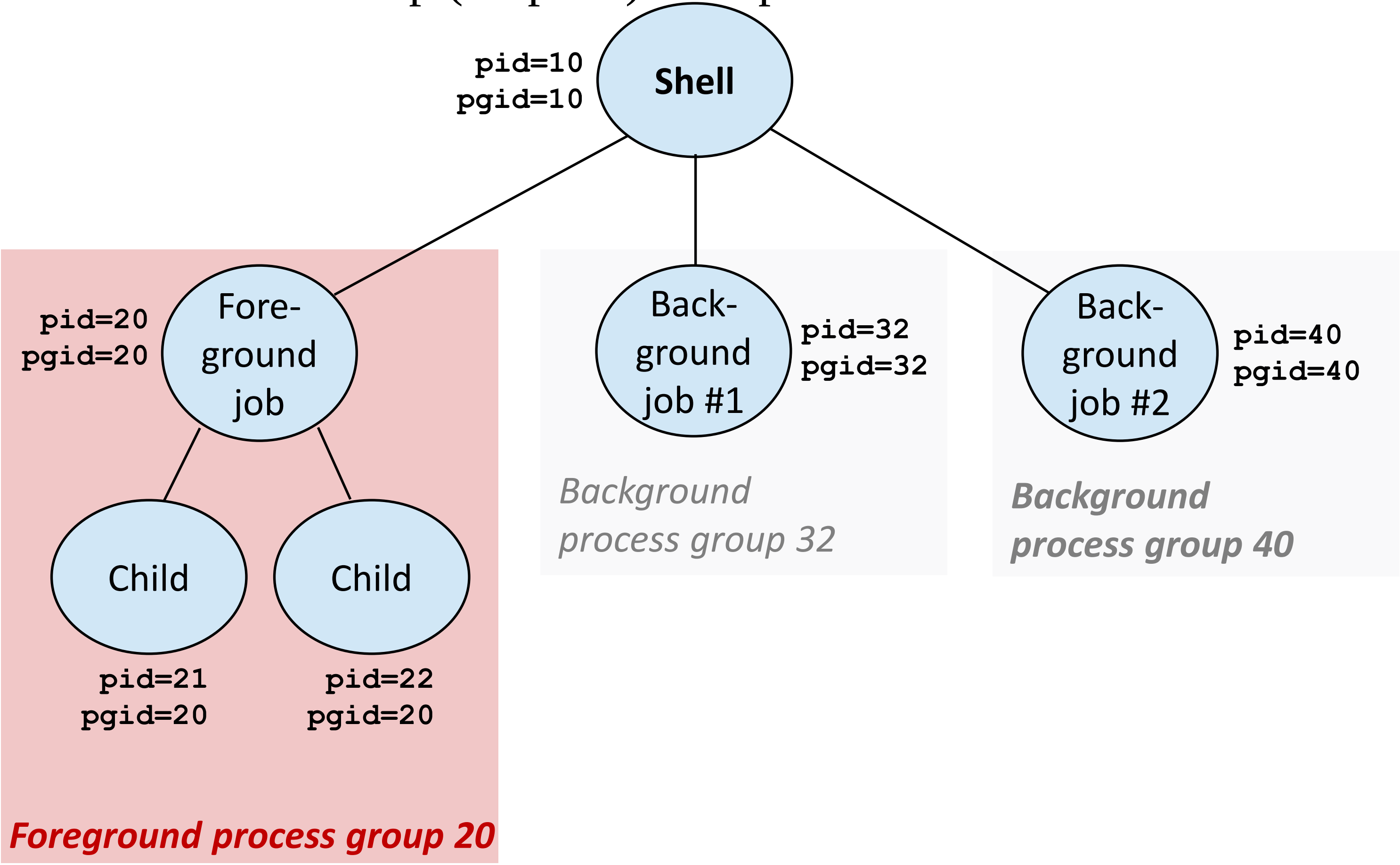
```
linux> ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps
linux> /bin/kill -9 -24817
linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
linux>
```

SENDING SIGNALS FROM THE KEYBOARD

- Typing ctrl-c (ctrl-z) causes the kernel to send a SIGINT (SIGTSTP) to every job in the foreground process group.
  - SIGINT – default action is to terminate each process
  - SIGTSTP – default action is to stop (suspend) each process

-9 *20



RECEIVING SIGNALS

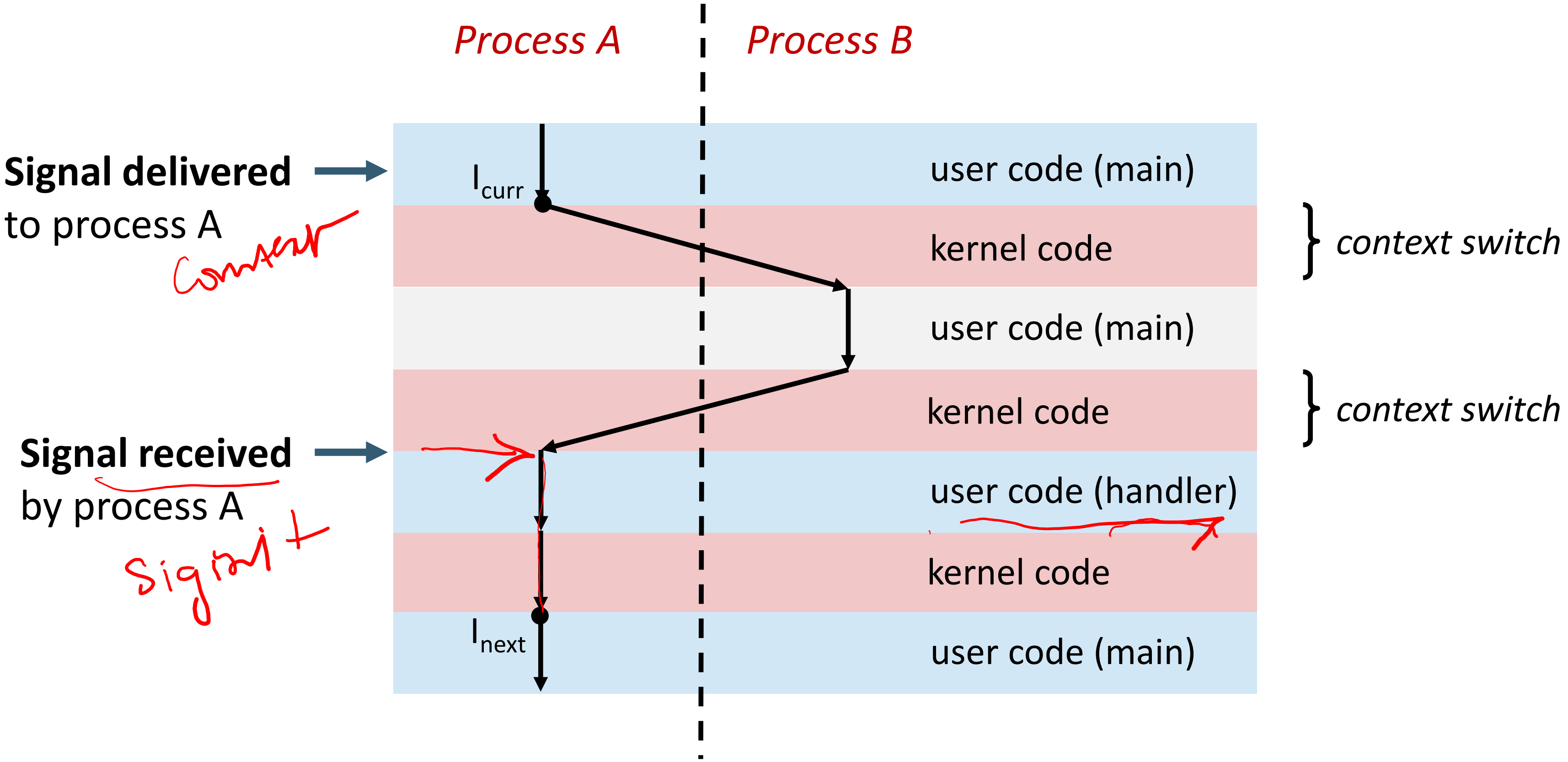
- Suppose kernel is returning from an exception handler and is ready to pass control to process *p*
- Kernel computes pnb = pending & ~blocked
  - The set of pending nonblocked signals for process *p*
- If (pnb == 0)
  - Pass control to next instruction in the logical flow for *p*
- Else
  - Choose least nonzero bit *k* in **pnb** and force process *p* to ***receive*** signal *k*
  - The receipt of the signal triggers some ***action*** by *p*
  - Repeat for all nonzero *k* in **pnb**
  - Pass control to next instruction in logical flow for *p*

Number	Name	Default action	Corresponding event
1	SIGHUP	Terminate	Terminal line hangup
2	SIGINT	Terminate	Interrupt from keyboard
3	SIGQUIT	Terminate	Quit from keyboard
4	SIGILL	Terminate	Illegal instruction
5	SIGTRAP	Terminate and dump core [a]	Trace trap
6	SIGABRT	Terminate and dump core [a]	Abort signal from `abort` function
7	SIGBUS	Terminate	Bus error
8	SIGFPE	Terminate and dump core [a]	Floating-point exception
9	SIGKILL	Terminate [b]	Kill program
10	SIGUSR1	Terminate	User-defined signal 1
11	SIGSEGV	Terminate and dump core [a]	Invalid memory reference (seg fault)
12	SIGUSR2	Terminate	User-defined signal 2
13	SIGPIPE	Terminate	Wrote to a pipe with no reader
14	SIGALRM	Terminate	Timer signal from `alarm` function
15	SIGTERM	Terminate	Software termination signal
16	SIGSTKFLT	Terminate	Stack fault on coprocessor
17	SIGCHLD	Ignore	A child process has stopped or terminated
18	SIGCONT	Ignore	Continue process if stopped
19	SIGSTOP	Stop until next SIGCONT [b]	Stop signal not from terminal
20	SIGTSTP	Stop until next SIGCONT	Stop signal from terminal
21	SIGTTIN	Stop until next SIGCONT	Background process read from terminal
22	SIGTTOU	Stop until next SIGCONT	Background process wrote to terminal
23	SIGURG	Ignore	Urgent condition on socket
24	SIGXCPU	Terminate	CPU time limit exceeded
25	SIGXFSZ	Terminate	File size limit exceeded
26	SIGVTALRM	Terminate	Virtual timer expired
27	SIGPROF	Terminate	Profiling timer expired
28	SIGWINCH	Ignore	Window size changed
29	SIGIO	Terminate	I/O now possible on a descriptor
30	SIGPWR	Terminate	Power failure

DEFAULT ACTIONS

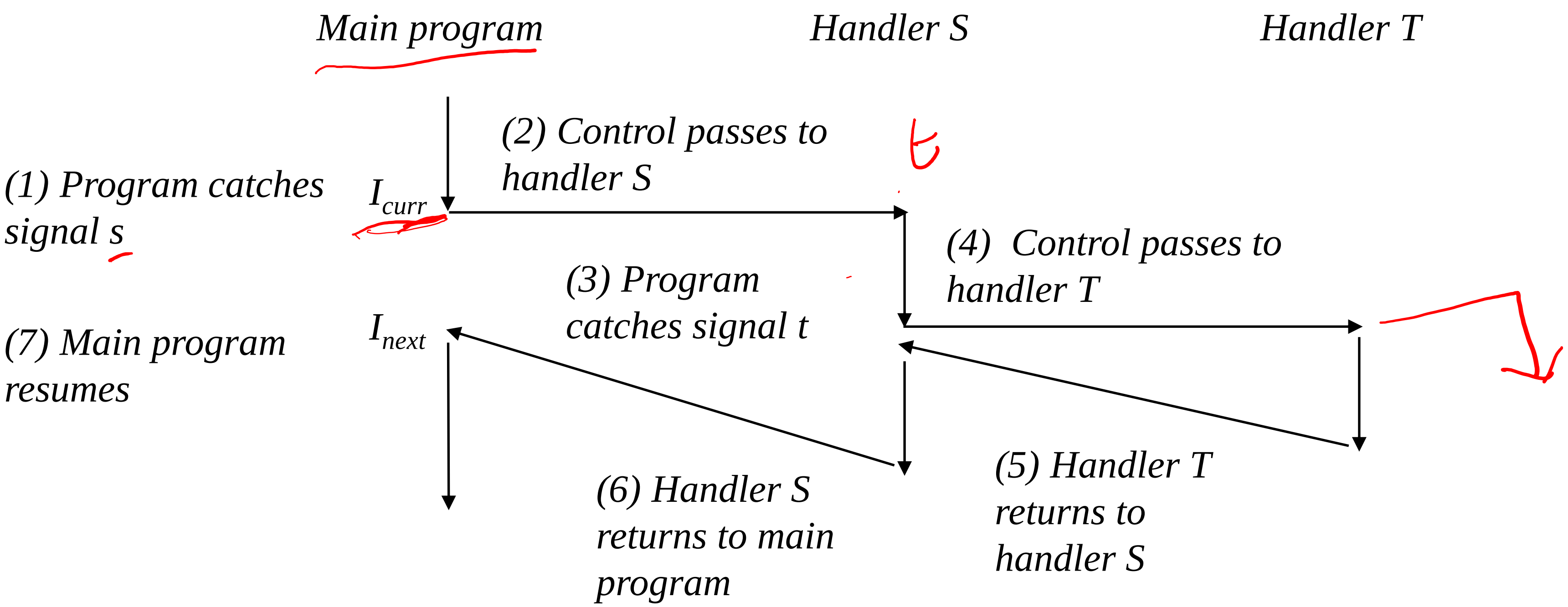
- Each signal type has a predefined *default action*, which is one of:
  - The process terminates
  - The process stops until restarted by a SIGCONT signal
  - The process ignores the signal

VIEW OF SIGNAL HANDLERS AS CONCURRENT FLOWS



NESTED SIGNAL HANDLERS

- Handlers can be interrupted by other handlers



BLOCKING AND UNBLOCKING SIGNALS

- Implicit blocking mechanism
  - Kernel blocks any pending signals of type currently being handled.
  - E.g., A SIGINT handler can't be interrupted by another SIGINT
- Explicit blocking and unblocking mechanism
  - sigprocmask function
- Supporting functions
  - sigemptyset – Create empty set
  - sigfillset – Add every signal number to set
  - sigaddset – Add signal number to set
  - sigdelset – Delete signal number from set

sigismember ( )

SAFE SIGNAL HANDLING

- Handlers are tricky because they are concurrent with main program and share the same global data structures.- interfere main and other handlers
- How and when signals are received is often counter intuitive
- Different system can have different signal handling semantics

Gulidelines

1. Keep handlers as simple as possible – global flag
2. Call async-signal –safe function in handlers
3. Save and restore errno
4. Protect accesses to shared global data structures by blocking all signals
5. Declared global variables with volatile
6. Declare flags with sig_atomic_t

PORTABLE SIGNAL HANDLING

- Ugh! Different versions of Unix can have different signal handling semantics
  - Some older systems restore action to default after catching signal
  - Some interrupted system calls can return with errno == EINTR
  - Some systems don’t block signals of the type being handled
- Solution: sigaction

```
handler_t *Signal(int signum, handler_t *handler)
{
   struct sigaction action, old_action;

   action.sa_handler = handler;
   sigemptyset(&action.sa_mask); /* Block sigs of type being handled */
   action.sa_flags = SA_RESTART; /* Restart syscalls if possible */

   if (sigaction(signum, &action, &old_action) < 0)
      unix_error("Signal error");
   return (old_action.sa_handler);
}
```

SYNCHRONIZING FLOWS TO AVOID NASTY CONCURRENCY BUGS

SIGCHLD

Parent

fork()

add add()

Child

zombie

global Jobs list

add

Remove()

P1

- Parent executes fork function – kernel runs the child
- The child terminates and becomes zombie- delivers SIGCHILD signal to parent
- Before parents gets executed the pending signal is requested by the kernel to be received by parent process
- The signal handler calls deletejob
- After the handler is complete, the kernel runs the parent process and adds child to the job list using addjob

Synchronisation Error- Race

Solution : block SIGCHILD signal

SUMMARY OF SIGNALS

- Signals provide process-level exception handling
  - Can generate from user programs
  - Can define effect by declaring signal handler
  - Be very careful when writing signal handlers

- Nonlocal jumps provide exceptional control flow within process
  - Within constraints of stack discipline