[CSED211] Introduction to Computer Software Systems

Lecture 14: Exceptional Control Flow - Signals

Prof. Jisung Park

CAOS
COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.11.29

ECF Exists at All Levels of a System

- Exceptions
  - Hardware and operating system kernel software
- Process context switch
  - Hardware timer and kernel software

Previous Lecture

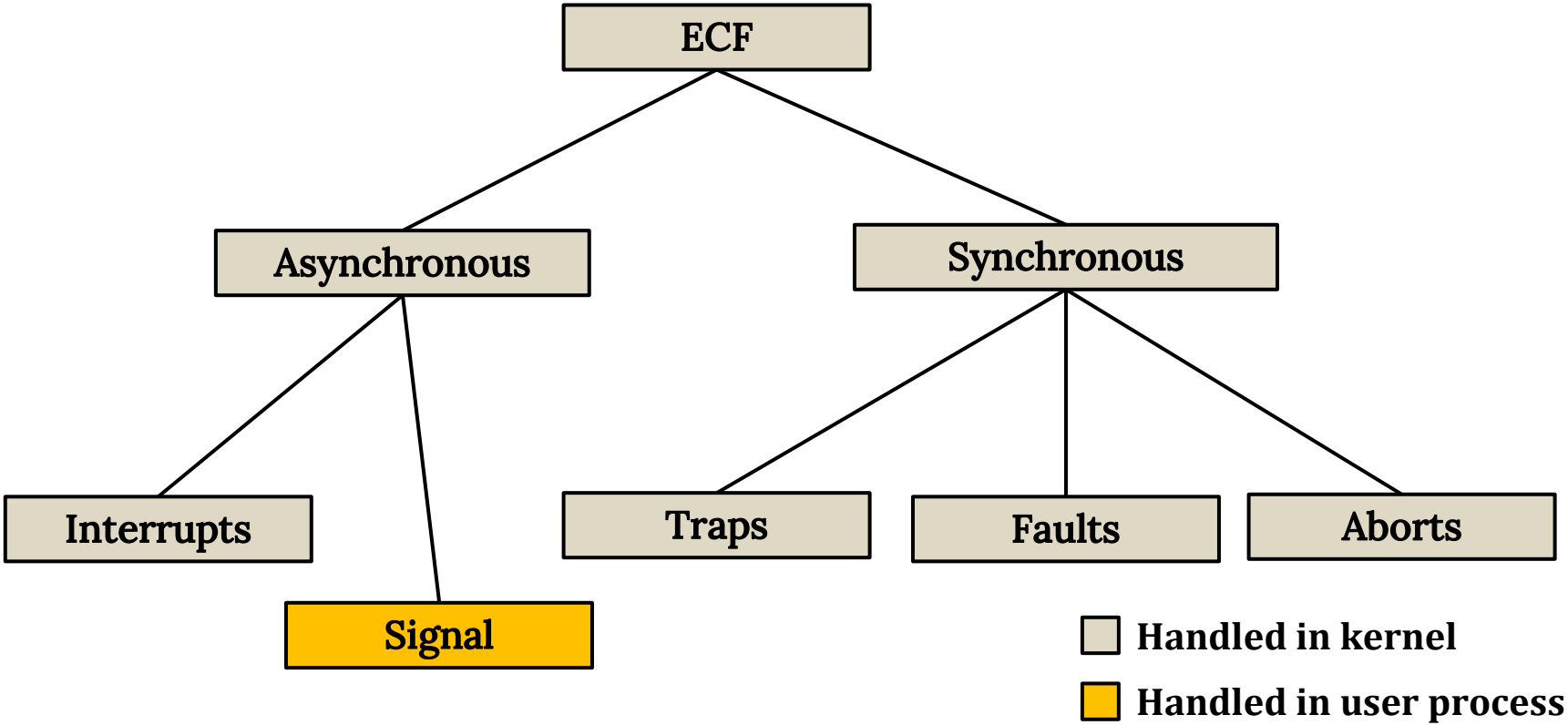
- Signals
  - Kernel software

This Lecture

- Nonlocal jumps
  - Application code

Textbook

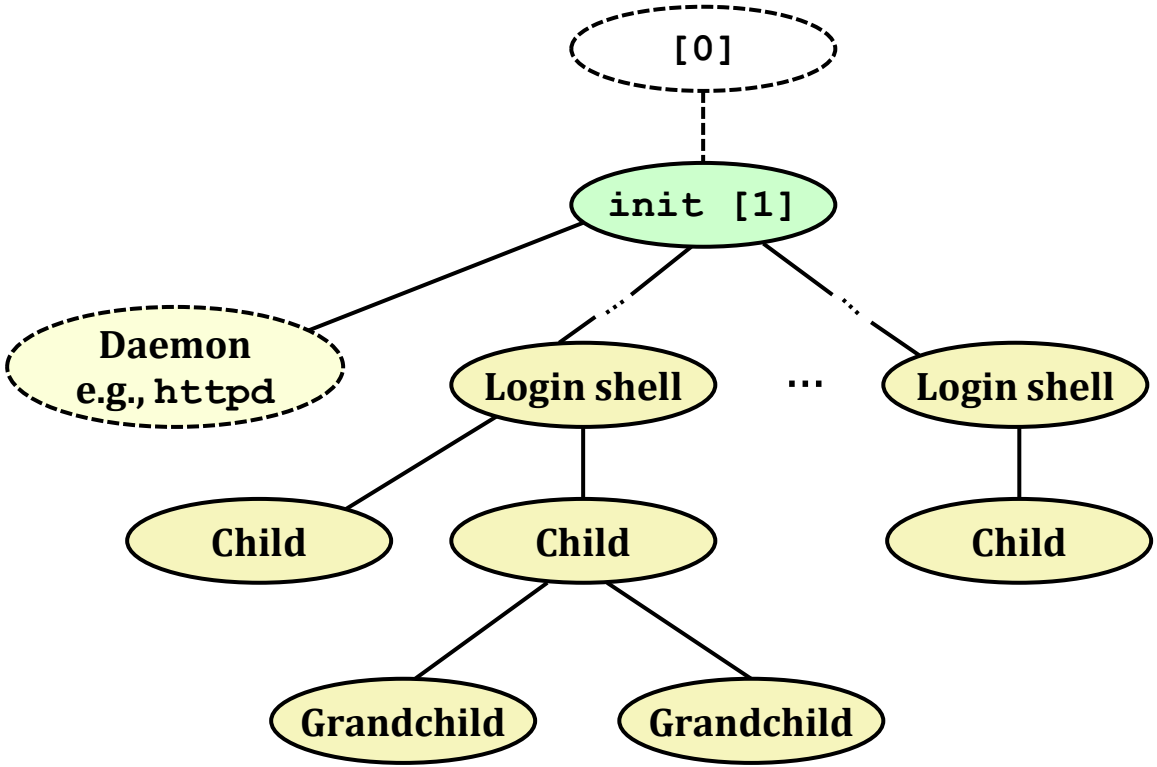
Taxonomy


ECF
Asynchronous
Synchronous
Interrupts
Signal
Traps
Faults
Aborts
Handled in kernel
Handled in user process

Lecture Agenda

- Shells
- Signals

Linux Process Hierarchy



Note: you can view the hierarchy using the Linux `pstree` command

Shell Programs

- Shell: an application program that runs programs on behalf of the user
  - **`sh`**: original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - **`csh/tcsh`**: BSD Unix C shell
  - **`bash`**: Bourne-Again shell (default Linux shell)

- Simple shell
  - Described in the textbook (pp. 753~)
  - Implementation of a very elementary shell
  - Purpose
    - Understand what happens when you type commands
    - Understand use and operation of process control operations

Simple Shell Example

```
$ ./shellex
> /bin/ls -l csapp.c
-rw-r--r-- 1 jspark users 23053 Jun 15  2023 csapp.c
> /bin/ps
  PID TTY          TIME CMD
31542 pts/2    00:00:01 tcsh
32017 pts/2    00:00:00 shellex
32019 pts/2    00:00:00 ps
> /bin/sleep 10 &
32031 /bin/sleep 10 &
> /bin/ps
 PID TTY          TIME CMD
31542 pts/2    00:00:01 tcsh
32024 pts/2    00:00:00 emacs
32030 pts/2    00:00:00 shellex
32031 pts/2    00:00:00 sleep
32033 pts/2    00:00:00 ps
> quit
```

Must give full pathnames for programs

Run program in background

Simple Shell Implementation

- Basic loop
  - Read line from command line
  - Execute the requested operation
    - Built-in command
      (only one implemented is **`quit`**)
    - Load and execute program from a file
  - Execution is a sequence of read & evaluation

```
int main(int argc, char** argv){
  /* command line */
  char cmdline[MAXLINE];

  while(1){
    /* read */
    printf("> ");
    fgets(cmdline, MAXLINE, stdin);
    if(feof(stdin))
      exit(0);

    /* evaluate */
    eval(cmdline);
  }
    …
```

shellex.c

Simple Shell Implementation

```
void eval(char *cmdline){
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if(argv[0] == NULL)
        return;   /* Ignore empty lines */

    if(!builtin_command(argv)){
        if((pid = Fork()) == 0){   /* Child runs user job */
            if(execve(argv[0], argv, environ) < 0){
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }
```

parseline parses 'buf' into 'argv' and returns whether or not input line ended in '&'

shellex.c

Simple Shell Implementation

```
void eval(char *cmdline){
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if(argv[0] == NULL)
        return;   /* Ignore empty lines */

    if(!builtin_command(argv)){
        if((pid = Fork()) == 0){    /* Child runs user job */
            if(execve(argv[0], argv, environ) < 0){
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }
```

If the target program is a built-in command, then handle it here. Otherwise, `fork/exec` the program

shellex.c

Simple Shell Implementation

```
    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if(argv[0] == NULL)
        return;   /* Ignore empty lines */

    if(!builtin_command(argv)){
        if((pid = Fork()) == 0){   /* Child runs user job */
            if(execve(argv[0], argv, environ) < 0){
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        /* Parent waits for foreground job to terminate */
        if(!bg){
            int status;
            if(waitpid(pid, &status, 0) < 0)
```

If the target program is a built-in command, then handle it here. Otherwise, `fork/exec` the program

Creat a child via `fork`

Execute the target program (specified by `argv[0]`)

Note: execve returns only on error

shellex.c

Simple Shell Implementation

```
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        /* Parent waits for foreground job to terminate */
        if(!bg){
            int status;
            if(waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}
```

If the program runs in foreground, the shell must wait for its completion

If the program runs in background, print its PID and continue doing other stuff

Is everything okay?

shellex.c

What Is a 'Background Job'?

- Users generally run one command at a time
  - Type command, read output, type another command

- Some programs run for a long time
  - e.g., delete this after two hours

```
$ sleep 7200; rm /tmp/junk
```

Shell stuck for 2 hours

- A background job is a process we do not want to wait for

```
$ (sleep 7200; rm /tmp/junk) &
[1] 907
$
```

Ready for the next command

Problem with the Simple Shell Example

- The shell program is designed to run indefinitely
  - i.e., until the user input is '`quit`'
  - Should not accumulate resources unnecessarily
    - Memory
    - Child processes
    - File descriptors

- Our example shell correctly waits for and reaps foreground jobs

- What about background jobs?
  - Will become zombies when they terminate
  - Will never be reaped because shell (typically) will not terminate
  - Will create a memory leak that could run the kernel out of memory

ECF to the Rescue

- Problem
  - The shell does not know when a background job will finish
    - Which could happen at any time by nature
  - The shell's regular control flow cannot reap exited background processes in a timely fashion
    - Regular control flow: wait until running job completes, then reap it

- Solution: exceptional control flow
  - The kernel interrupts regular processing to alert the shell (parent) when a background process (child) completes
  - Such an alert mechanism is called a signal in UNIX

Lecture Agenda

- Shells
- Signals

Signal

- A small message that notifies a process of a system event
  - Akin to exceptions and interrupts
  - Sent from the kernel to a process (sometimes at the request from another process)
  - Signal type is identified by small integer ID's (1~30)
  - The only information in a signal is its ID (and its arrival for sure)

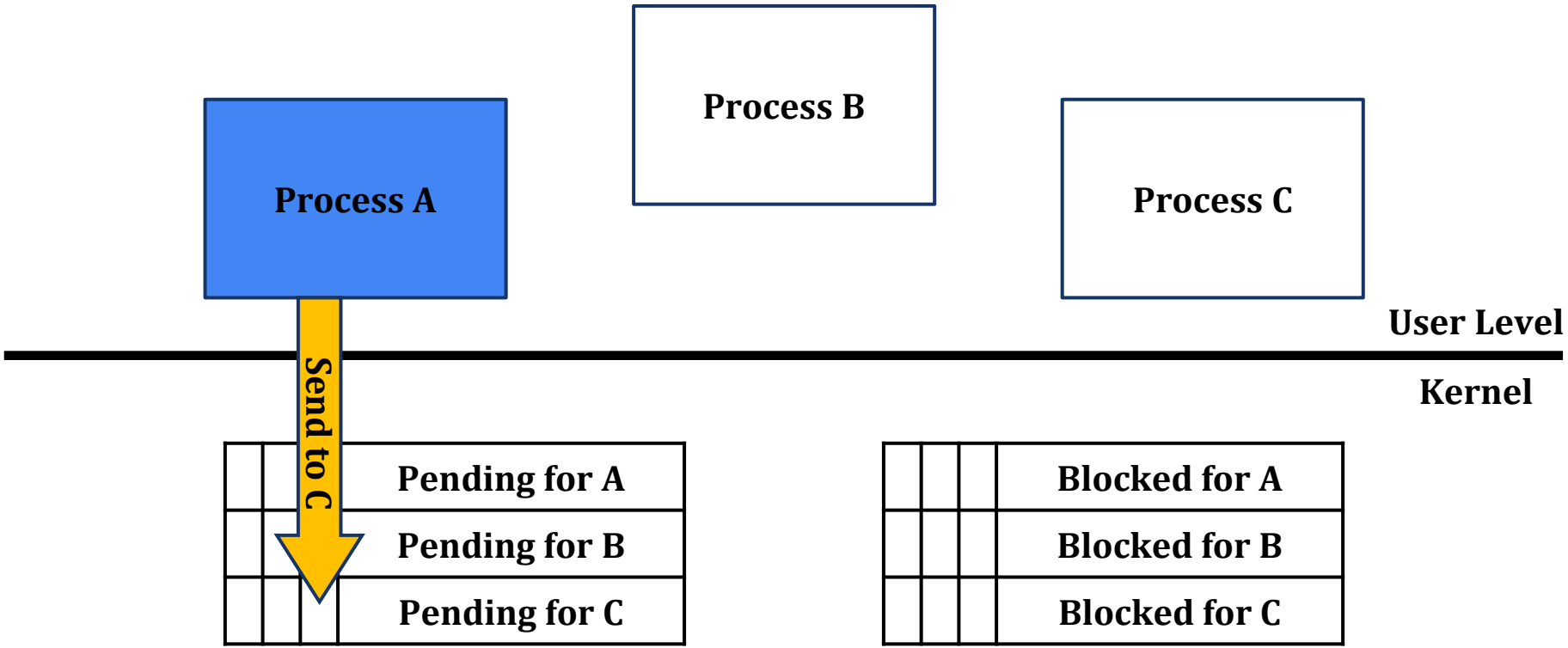
ID	Name	Default Action	Corresponding Event
2	`SIGINT`	Terminate	Interrupt (e.g., CTRL-C from keyboard)
9	`SIGKILL`	Terminate	Kill program (cannot override or ignore)
11	`SIGSEGV`	Terminate & Dump	Segmentation violation
14	`SIGALRM`	Terminate	Timer signal
17	`SIGCHLD`	Ignore	Child stopped or terminated

Signal Concepts: Sending a Signal

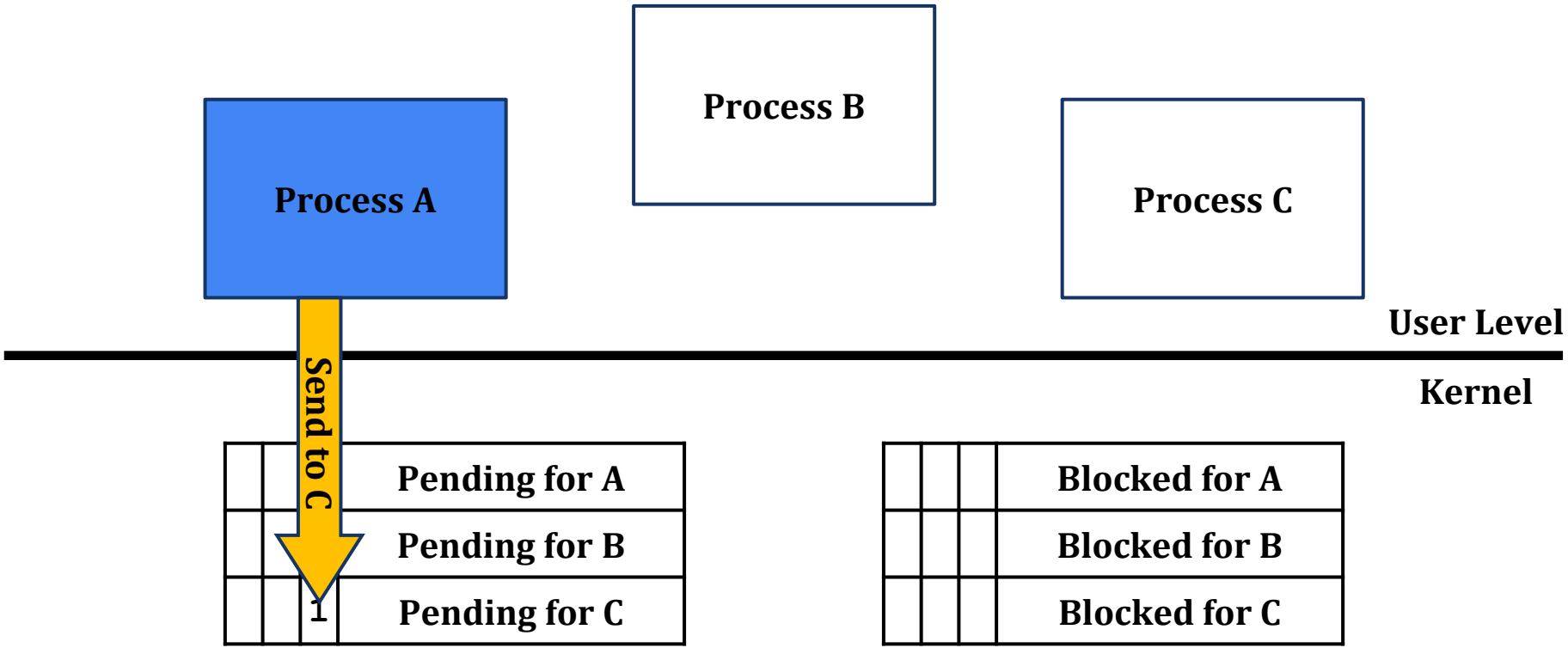
- Kernel sends (delivers) a signal to a destination process by updating some state in the context of the destination process

- 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

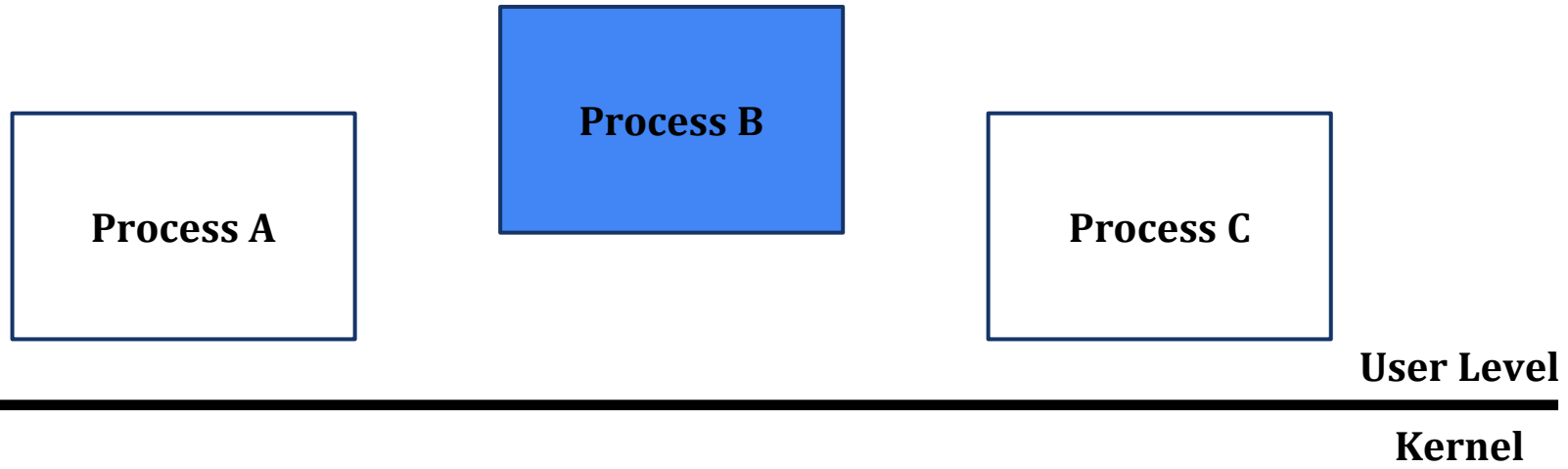
Signal Concepts: Sending a Signal (Cont.)



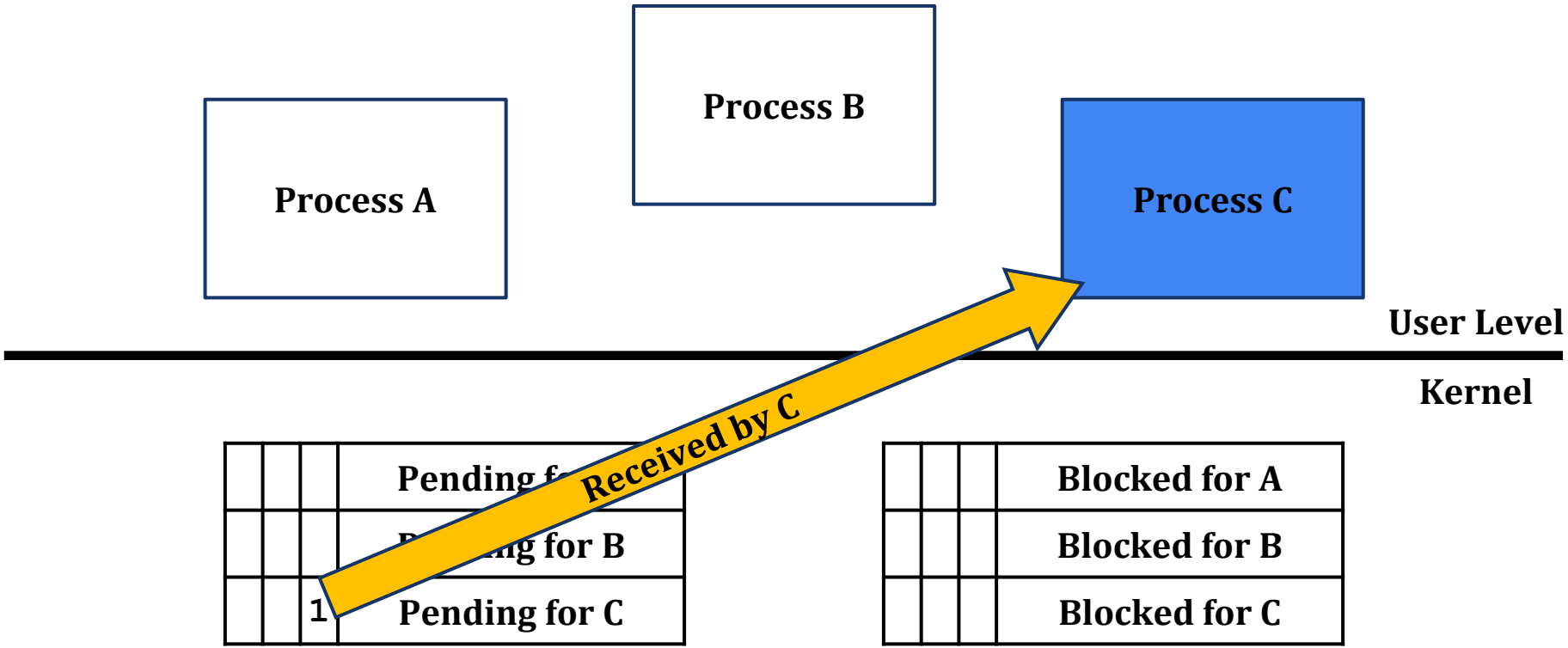
Signal Concepts: Sending a Signal (Cont.)



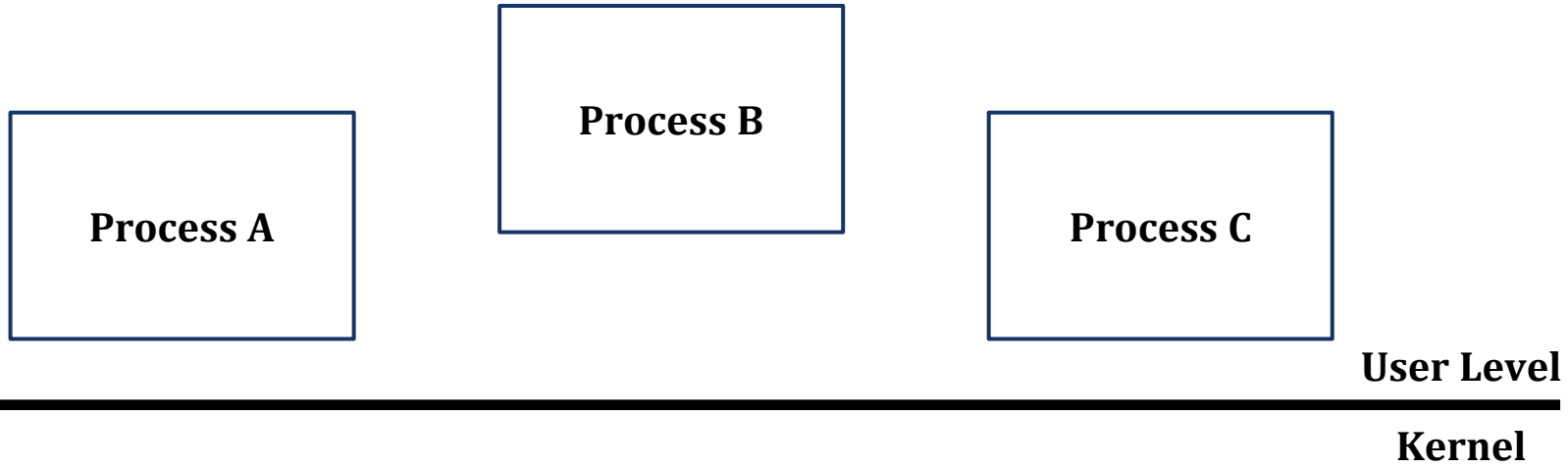
Signal Concepts: Sending a Signal (Cont.)



Signal Concepts: Sending a Signal (Cont.)



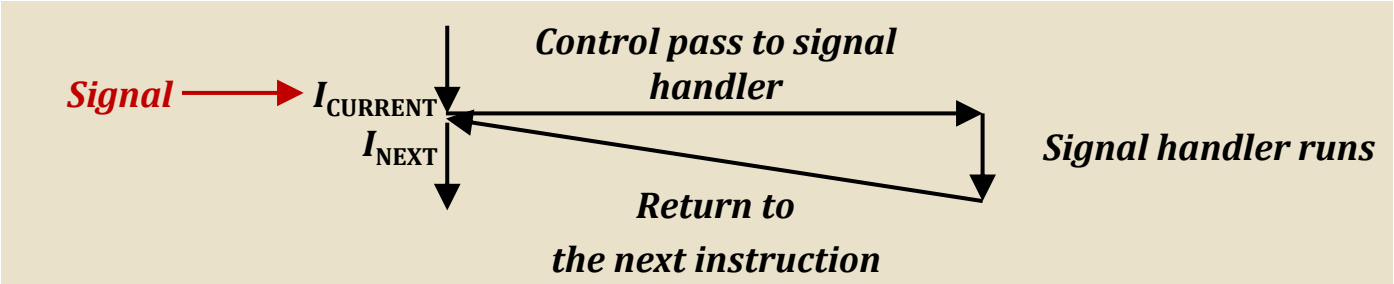
Signal Concepts: Sending a Signal (Cont.)



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

- Three 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

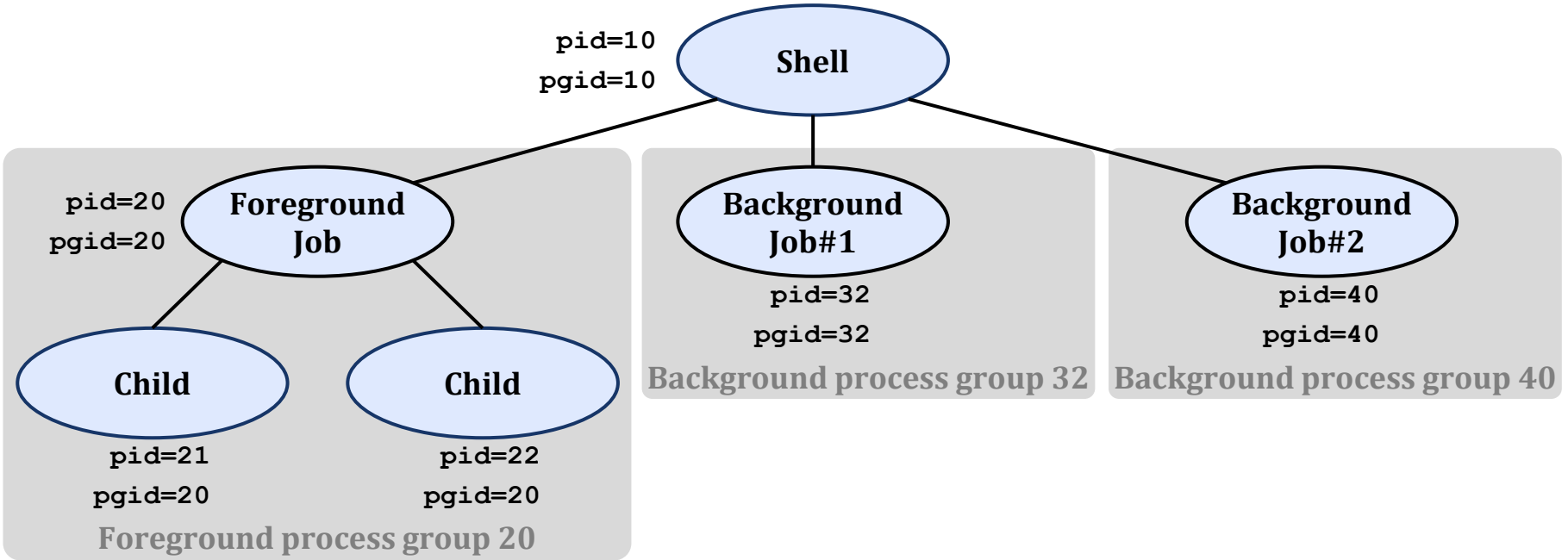


Signal Concepts: Pending and Blocked Signals

- 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 type-*k* signals will be discarded
  - A pending signal is received at most once
- A process can block the receipt of certain signals
  - A blocked signal can be delivered, but will not be received until unblocked
- Kernel keeps bit vectors **`pending/blocked`** in the context of each process
  - Represent the sets of pending/blocked signals
  - Kernel sets (clears) bit *k* in **`pending`** when a signal of type *k* is delivered (received)
  - A process can (un)block a signal of type *k* by setting (clearing) bit *k* in **`blocked`** (also referred to as the signal mask) using the **`sigprocmask`** function

Sending Signals: Process Groups

- Every process belongs to exactly one process group
  - `setpgid()`: changes the process group of a process
  - `getpgrp()`: returns the current process group



Sending Signals with `/bin/kill` Program

- Sends arbitrary signal to a process or process group
- Example

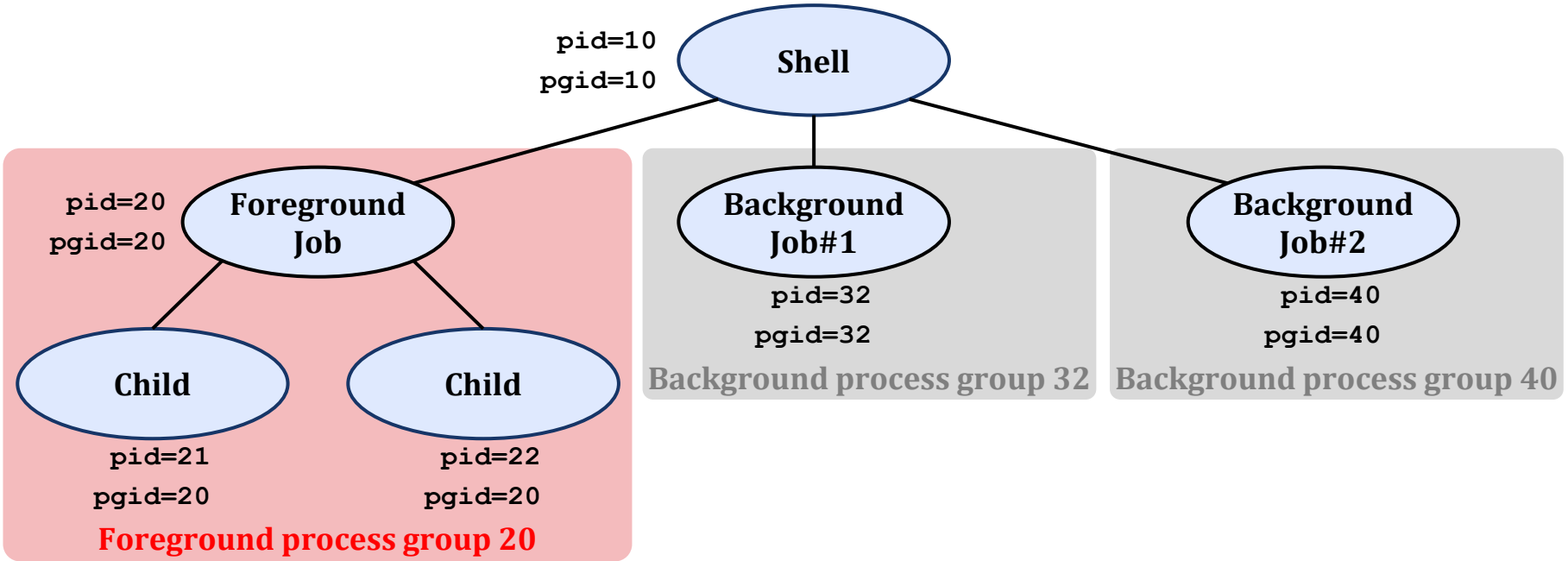
```
$ ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817
$ ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps
$ /bin/kill -9 -24817
$ ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
$
```

`/bin/kill -9 24818`: sends `SIGKILL` to process 24818

`/bin/kill -9 24817`: sends `SIGKILL` to every process in process group 24817

Sending Signals with Keyboard

- CTRL-C (CTRL-Z) sends a **SIGINT** (**SIGTSTP**) to every job in the foreground process group; default action is to terminate (suspend) each process



Sending Signals with Keyboard: Example

```
$ ./forks 17
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107
Suspended
$ ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28107 pts/8    T      0:01 ./forks 17
28108 pts/8    T      0:01 ./forks 17
28109 pts/8    R+     0:00 ps w
$ fg
./forks 17
$ ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28110 pts/8    R+     0:00 ps w
```

Types CTRL-Z

Types CTRL-C

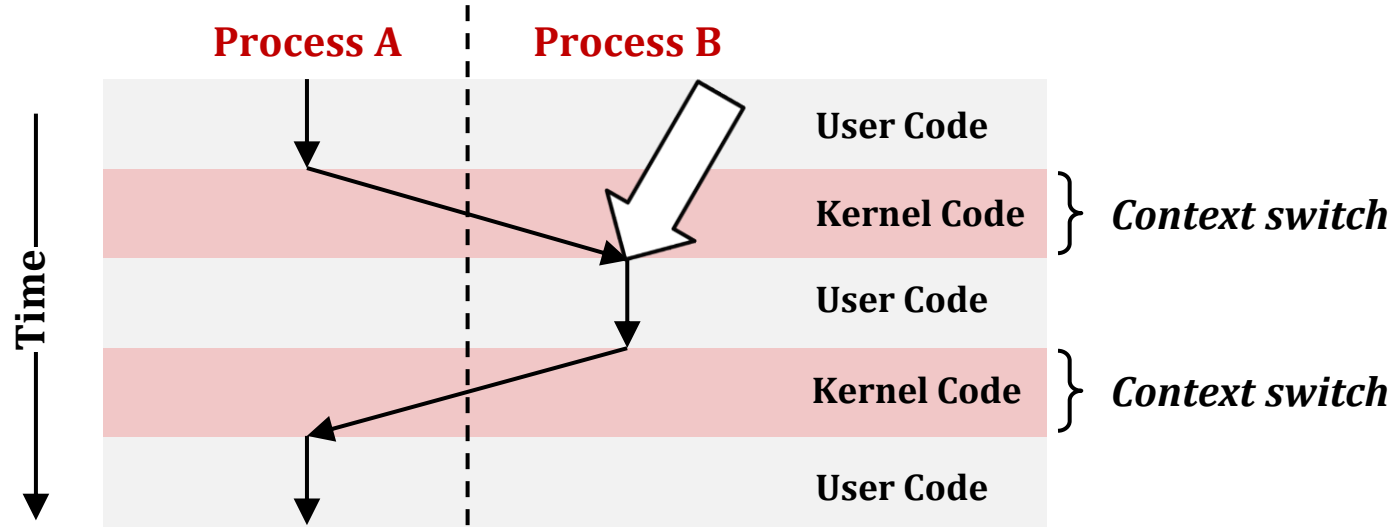
- STAT (process state) legend
  - First letter
    - **S**: sleeping
    - **T**: stopped
    - **R**: running
  - Second letter
    - **s**: session leader
    - **+**: foreground proc. group
  - See '**man ps**' for more details

Sending Signals with `kill` Function

```
void fork12(){
  pid_t pid[N];
  int i, child_status;
  for(i = 0; i < N; i++)
    if((pid[i] = fork()) == 0) while(1); /* Child infinite loop */

  /* Parent terminates the child processes */
  for(i = 0; i < N; i++){
    printf("Killing process %d\n", pid[i]);
    kill(pid[i], SIGINT);
  }

  /* Parent reaps terminated children */
  for(i = 0; i < N; i++){
    pid_t wpid = wait(&child_status);
    if(WIFEXITED(child_status))
      printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
    else
      printf("Child %d terminated abnormally\n", wpid);
  }
}
```

Receiving Signals

- Suppose kernel is returning from an exception handler and is ready to pass control to process p



Receiving Signals

- Suppose kernel is returning from an exception handler and is ready to pass control to process p

- Kernel computes `pnb = pending & ~blocked`
  - i.e., the set of pending non-blocked signals for process p

- If (`pnb == 0`): pass control to the next instruction in the logical flow for p

- Otherwise
  - Choose the nonzero LSB k in `pnb` and force process p to receive signal k
  - The receipt of the signal triggers the corresponding action by p
  - Repeat for all nonzero bits in `pnb`
  - Pass control to the next instruction in logical flow for p

Default Actions

- Each signal type has a predefined default action that is one of
  - The process terminates
  - The process terminates and dumps core
  - The process stops until restarted by a `SIGCONT` signal
  - The process ignores the signal

Installing Signal Handlers

- The **`signal`** function modifies the default action for signal **`signum`**
  - **`handler_t *signal(int signum, hander_t *handler)`**

- Different values for **`handler`**
  - **`SIG_IGN`**: ignore signals of type **`signum`**
  - **`SIG_DFL`**: revert to the default action on receipt of signals of type **`signum`**
  - Otherwise, **`handler`** is the address of a signal handler
    - Called when process receives signal of type **`signum`**
    - Referred to as installing the handler
    - Executing the handler is called catching or handling the signal
    - When the handler executes its return statement, control passes back to the instruction in the control flow of the process that was interrupted by receipt of the signal

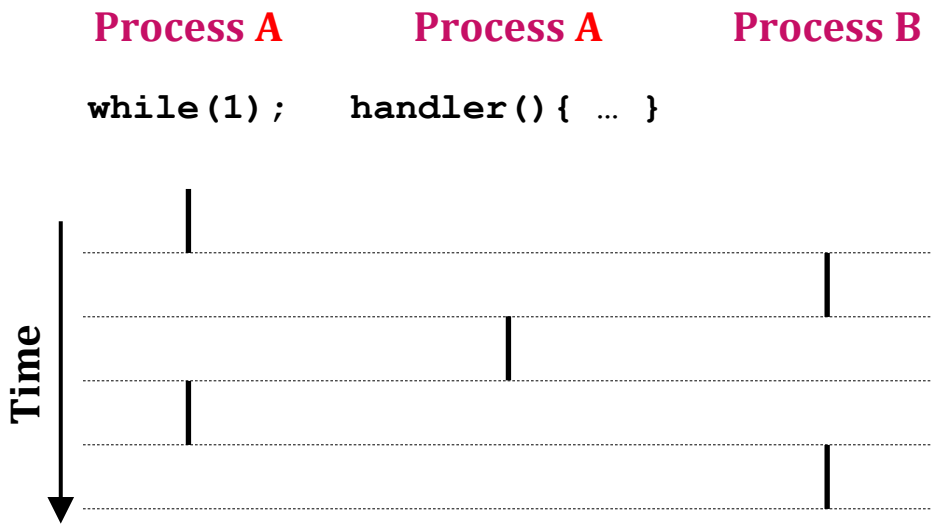
Signal Handling Example

```c
/* SIGINT handler */
void sigint_handler(int sig){
    printf("So you think you can stop the bomb with ctrl-c, do you?\n");
    sleep(2);
    printf("Well...");
    fflush(stdout);
    sleep(1);
    printf("OK. :-)\n");
    exit(0);
}

int main(){
    /* Install the SIGINT handler */
    if(signal(SIGINT, sigint_handler) == SIG_ERR)
        unix_error("signal error");

    /* Wait for the receipt of a signal */
    pause();
    return 0;
}
```

sigint.c

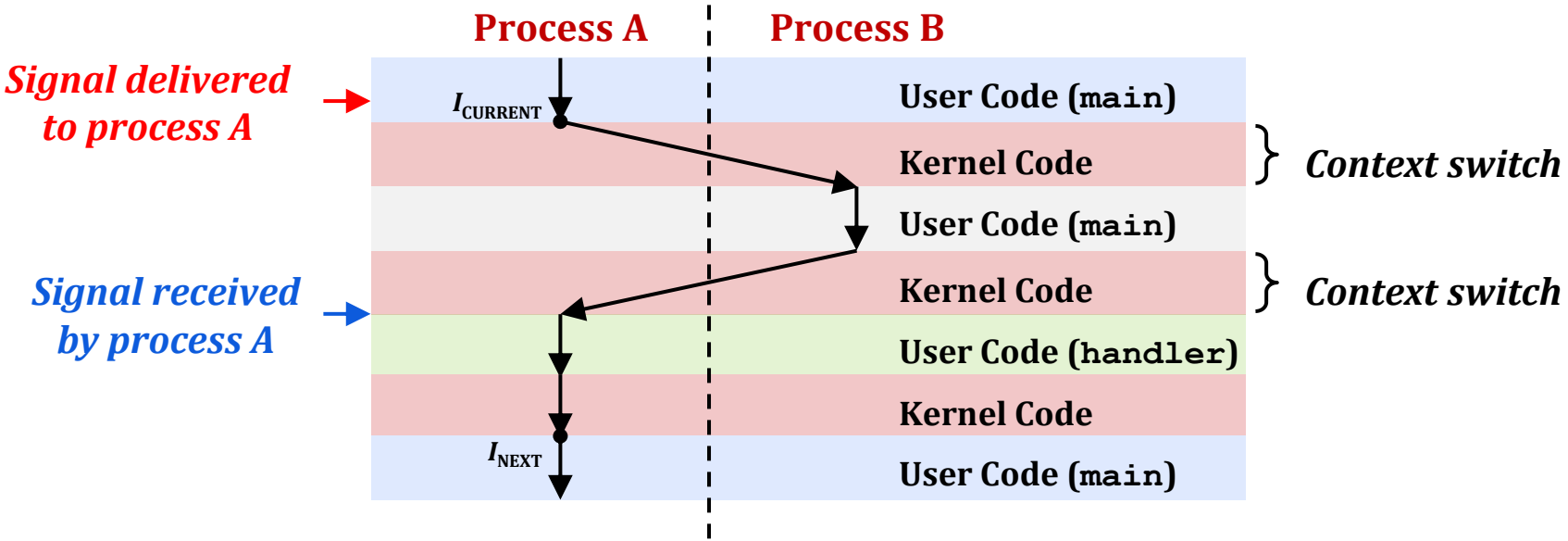
Signal Handlers as Concurrent Flows

- A signal handler is a separate logical flow (not process)
  - Runs concurrently with the main program
  - Exists only until returns to the main program



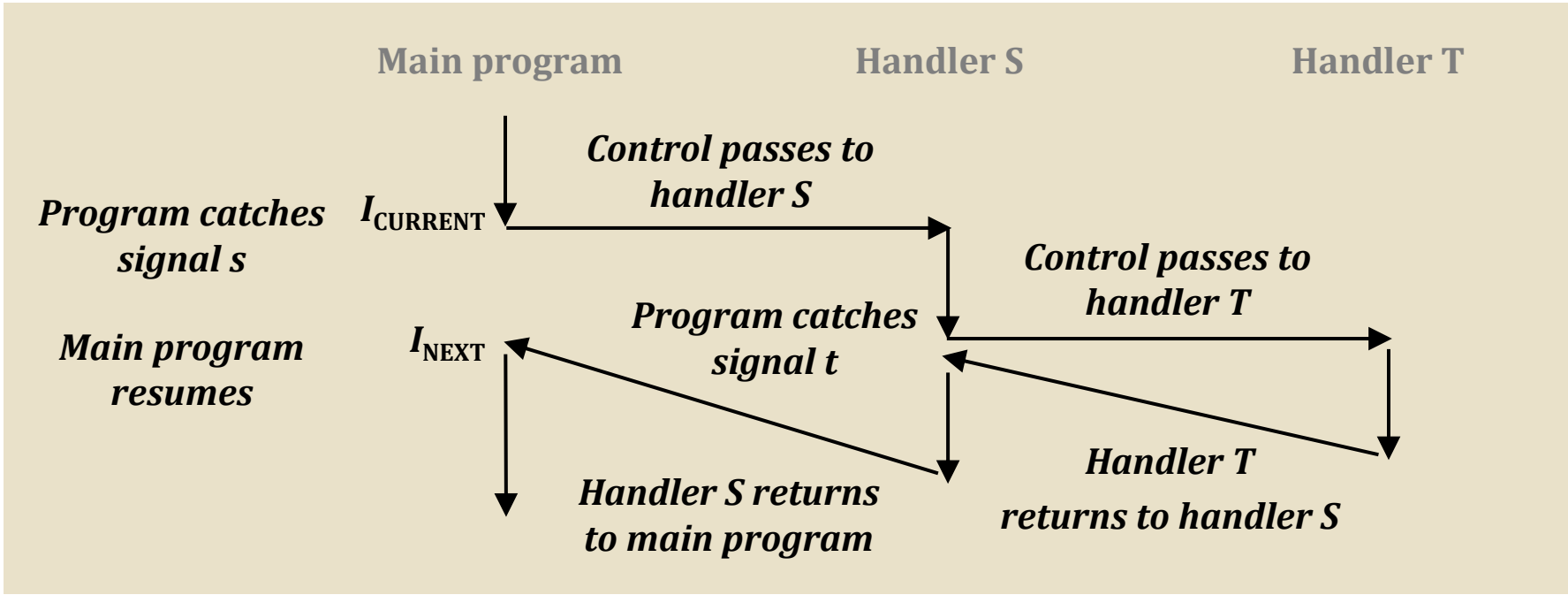
Signal Handlers as Concurrent Flows: Another View

- A signal handler is a separate logical flow (not process)
  - Runs concurrently with the main program
  - Exists only until returns to the main program



Nested Signal Handlers

- Handlers can be interrupted by other handlers



Blocking and Unblocking Signals

- Implicit blocking mechanism
  - Kernel blocks any pending signals of the type currently being handled
    e.g., a **`SIGINT`** handler cannot be interrupted by another **`SIGINT`**

- Explicit blocking and unblocking mechanism: **`sigprocmask`** function

- Supporting functions
  - **`sigemptyset`**: create an empty set
  - **`sigfillset`**: add every signal number to set
  - **`sigaddset`**: add signal number to set
  - **`sigdelset`**: delete signal number from set

Temporarily Blocking Signals

```
sigset_t mask, prev_mask;

sigemptyset(&mask);
sigaddset(&mask, SIGINT);

/* Block SIGINT and save previous blocked set */
sigprocmask(SIG_BLOCK, &mask, &prev_mask);

/* Code region that will not be interrupted by SIGINT */

/* Restore previous blocked set, unblocking SIGINT */
sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

Safe Signal Handling

- Handlers are tricky because they are concurrent with main program and share the same global data structures
  - Shared data structures can become corrupted

- We will (hopefully) discuss concurrency issues later in this course; for now, here are some guidelines to help you avoid trouble

Guildlines for Writing Safe Handlers

- G0: Keep your handlers as simple as possible
  - e.g., set a global flag and return
- G1: Call only async-signal-safe function in your handlers
  - `printf`, `sprintf`, `malloc`, and `exit` are not safe
- G2: Save and restore `errno` on entry and exit
  - So that other handlers do not overwrite your value of `errno`
- G3: Protect shared data structures by temporarily blocking all signals
- G4: Declare global variables as `volatile`
  - To prevent compiler from storing them in a register
- G5: Declare global flags (only read and written) as `volatile sig_atomic_t`
  - So that they do not need to be protected like other globals

Async-Signal-Safety

- A function is async-signal-safe if it meets either of the following conditions
  - Reentrant, i.e., all variables stored on stack frame (§12.7.2 in the textbook)
  - Non-interruptible by signals

- Posix guarantees 117 functions to be async-signal-safe
  - Source: **`man 7 signal`**
  - Popular functions on the list: **`_exit`**, **`write`**, **`wait`**, **`waitpid`**, **`sleep`**, **`kill`**
  - Popular functions not on the list
    - **`printf`**, **`sprintf`**, **`malloc`**, **`exit`**
    - Unfortunate fact: **`write`** is the only async-signal-safe output function

Safe Formatted Output Option#1

- Use the reentrant SIO (safe I/O library) from `csapp.c` in your handlers
  - `ssize_t sio_puts(char s[]) /* Put string */`
  - `ssize_t sio_putl(long v)   /* Put long */`
  - `void sio_error(char s[])   /* Put msg & exit */`

```
/* Safe SIGINT handler */
void sigint_handler(int sig){
    sio_puts("So you think you can stop the bomb with ctrl-c, do you?\n");
    sleep(2);
    sio_puts("Well...");
    sleep(1);
    sio_puts("OK. :-)\n");
    _exit(0);
}
```

Safe Formatted Output Option#2

- Use the new & improved reentrant **`sio_printf`**
  - Handles restricted class of **`printf`** format strings
    - Recognizes **`%c`**, **`%s`**, **`%d`**, **`%u`**, **`%x`**, and **`%%`**
    - Size designators **`l`** and **`z`**

```
/* Safe SIGINT handler */
void sigint_handler(int sig){
    sio_printf("So you think you can stop the bomb (process %d)
                with ctrl-%c, do you?\n", (int) getpid(), 'c');
    sleep(2);
    sio_puts("Well...");
    sleep(1);
    sio_puts("OK. :-)\n");
    _exit(0);
}
```

Correct Signal Handling

- Pending signals are not queued
  - Only one bit in the pending bit vector for each signal type
    → At most one pending signal of any particular type
  - You cannot use signals to count events, such as children terminating

```
volatile int ccount = 0;
void child_handler(int sig){
    int olderrno = errno;
    pid_t pid;
    if((pid = wait(NULL)) < 0)
        sio_error("wait error");
    ccount--;
    sio_puts("Handler reaped child ");
    sio_putl((long) pid);
    sio_puts(" \n");
    sleep(1);
    errno = olderrno;
}
```

```
void fork14(){
    pid_t pid[N];
    int i;
    ccount = N;
    signal(SIGCHLD, child_handler);

    for(i = 0; i < N; i++){
        if((pid[i] = Fork()) == 0){
            sleep(1);
            exit(0);  /* Child exits */
        }
    }
    while(ccount > 0); /* Parent spins */
}
```

Correct Signal Handling

- Pending signals are not queued
  - Only one bit in the pending bit vector for each signal type
    → At most one pending signal of any particular type
  - You cannot use signals to count events, such as children terminating

```
volatile int ccount = 0;
void child_handler(int sig){
    int olderrno = errno;
    pid_t pid;
    if((pid = wait(NULL)) < 0)
        sio_error("wait error");
    ccount--;
    sio_puts("Handler reaped child ");
    sio_putl((long) pid);
    sio_puts(" \n");
    sleep(1);
    errno = olderrno;
}
```

```
void fork14(){
    pid_t pid[N];
    int i;
    ccount = N;
    signal(SIGCHLD, child_handler);

    for(i = 0; i < N; i++){
        if((pid[i] = Fork()) == 0){
            sleep(1);
            exit(0);  /* Child exits */

unt > 0); /* Parent spins */
```

```
$ ./forks 14
Handler reaped child 23240
Handler reaped child 23241
```
Program hangs

Correct Signal Handling (Cont.)

- Must call wait for all terminated child processes
  - Put **`wait`** in a loop to reap all terminated children

```
$ ./forks 15
Handler reaped child 23246
Handler reaped child 23247
Handler reaped child 23248
Handler reaped child 23249
Handler reaped child 23250
$
```

```
volatile int ccount = 0;
void child_handler2(int sig){
    int olderrno = errno;
    pid_t pid;
    while((pid = wait(NULL)) > 0){
        ccount--;
        sio_puts("Handler reaped child ");
        sio_putl((long) pid);
        sio_puts(" \n");
    }
    if(errno != ECHILD)
        sio_error("wait error");
    errno = olderrno;
}
```

```
void fork15(){
    pid_t pid[N];
    int i;
    ccount = N;
    signal(SIGCHLD, child_handler);

    for(i = 0; i < N; i++){
        if((pid[i] = Fork()) == 0){
            sleep(1);
            exit(0);  /* Child exits */
        }
    }
    while(ccount > 0); /* Parent spins */
}
```

Synchronizing Flows to Avoid Races

- `SIGCHLD` handler for a simple shell
  - Blocks all signals while running critical code

```
void handler(int sig){
    int olderrno = errno;
    sigset_t mask_all, prev_all;
    pid_t pid;

    sigfillset(&mask_all);
    while((pid = waitpid(-1, NULL, 0)) > 0){
        /* Reap child */
        sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid); /* Delete the child from the job list */
        sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    if(errno != ECHILD)
        sio_error("waitpid error");
    errno = olderrno;
}
```

procmask1.c

Synchronizing Flows to Avoid Races (Cont.)

- Simple shell with a subtle synchronization error due to its wrong assumption that parent always runs before child

```
int main(int argc, char **argv){
    int pid;
    sigset_t mask_all, prev_all;
    int n = N;  /* N = 5 */
    sigfillset(&mask_all);
    signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */
    while(n--){
        if((pid = Fork()) == 0) /* Child */
            execve("/bin/date", argv, NULL);
        sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Parent */
        addjob(pid);  /* Add the child to the job list */
        sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    exit(0);
}
```

procmask1.c

Corrected Shell Program Without Race

```
int main(int argc, char **argv){
    int pid;
    sigset_t mask_all, mask_one, prev_one;
    int n = N;  /* N = 5 */
    sigfillset(&mask_all);
    sigemptyset(&mask_one);
    sigaddset(&mask_one, SIGCHLD);
    signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */
    while(n--){
        sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
        if((pid = Fork()) == 0){ /* Child */
            sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
            execve("/bin/date", argv, NULL);
        }
        sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Parent */
        addjob(pid);  /* Add the child to the job list */
        sigprocmask(SIG_SETMASK, &prev_one, NULL);
    }
    exit(0);
}
```

procmask2.c

Explicitly Waiting for Signals

- Handlers for program explicitly waiting for **`SIGCHILD`** to arrive

```c
volatile sig_atomic_t pid;

void sigchld_handler(int s){
    int olderrno = errno;
    pid = waitpid(-1, NULL, 0); /* Main is waiting for nonzero pid */
    errno = olderrno;
}

void sigint_handler(int s){
}
```

waitforsignal.c

Explicitly Waiting for Signals (Cont.)

```
int main(int argc, char **argv){
    sigset_t mask, prev;
    int n = N;  /* N = 10 */
    signal(SIGCHLD, sigchld_handler);
    signal(SIGINT, sigint_handler);
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);

    while(n--){
        sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
        if(Fork() == 0) /* Child */
            exit(0);
        pid = 0; /* Parent */
        sigprocmask(SIG_SETMASK, &prev, NULL); /* Unblock SIGCHLD */
        while(!pid); /* Wait for SIGCHLD to be received (wasteful!) */
        printf("."); /* Do something after receiving SIGCHLD */

    }
    printf("\n");
    exit(0);
}
```

Similar to a shell waiting for a foreground job to terminate

waitforsignal.c

Explicitly Waiting for Signals (Cont.)

- `while(!pid)`
  - Program is correct, but very wasteful
  - In a busy-wait loop

- `while(!pid) pause();`
  - Possible race condition
  - It might receive signal b/w checking `pid` and calling `pause()`

- `while(!pid) sleep();`
  - Safe, but slow
  - It will take up to one second to respond

Waiting for Signals with `sigsuspend`

- `int sigsuspend(const sigset_t *mask)`

- Equivalent to atomic (uninterruptable) version of

```
sigprocmask(SIG_BLOCK, &mask, &prev);
pause();
sigprocmask(SIG_SETMASK, &prev, NULL);
```

Waiting for Signals with `sigsuspend` (Cont.)

```
int main(int argc, char **argv){
    sigset_t mask, prev;
    int n = N;  /* N = 10 */
    signal(SIGCHLD, sigchld_handler);
    signal(SIGINT, sigint_handler);
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);

    while(n--){
        sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
        if(Fork() == 0) /* Child */
            exit(0);
        pid = 0; /* Parent */
        while(!pid) sigsuspend(&prev); /* Wait for SIGCHLD to be received */
        sigprocmask(SIG_SETMASK, &prev, NULL); /* Optionally unblock SIGCHLD */
        printf("."); /* Do something after receiving SIGCHLD */
    }
    printf("\n");
    exit(0);
}
```

sigsuspend.c

Lecture Agenda

- Shells
- Signals
- Portable Signal Handling
- Nonlocal Jumps

Consult your textbook!

Summary

- 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

[CSED211] Introduction to Computer Software Systems

Lecture 14: Exceptional Control Flow - Signals

Prof. Jisung Park

CAOS
COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.11.29