CSC 252/452: Computer Organization
Fall 2024: Lecture 19

Instructor: Yanan Guo

Department of Computer Science
University of Rochester

Signals

- A signal is a small message that notifies a process that an event of some type has occurred in the system
  - Sent from the **OS kernel**
  - Could be requested by another process, by user, or automatically by the kernel
  - Signal type is identified by small integer ID's (1-30)

Signals

- A signal is a small message that notifies a process that an event of some type has occurred in the system
  - Sent from the **OS kernel**
  - Could be requested by another process, by user, or automatically by the kernel
  - Signal type is identified by small integer ID's (1-30)

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

Default Actions to Signals

- 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

Installing Signal Handlers

- The signal function modifies the default action associated with the receipt of signal signum:
  - `handler_t *signal(int signum, handler_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 user-level function (signal handler)
    - Called when process receives signal of type signum
    - Referred to as "installing" the handler
    - Executing handler is called "catching" or "handling" the signal
    - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal

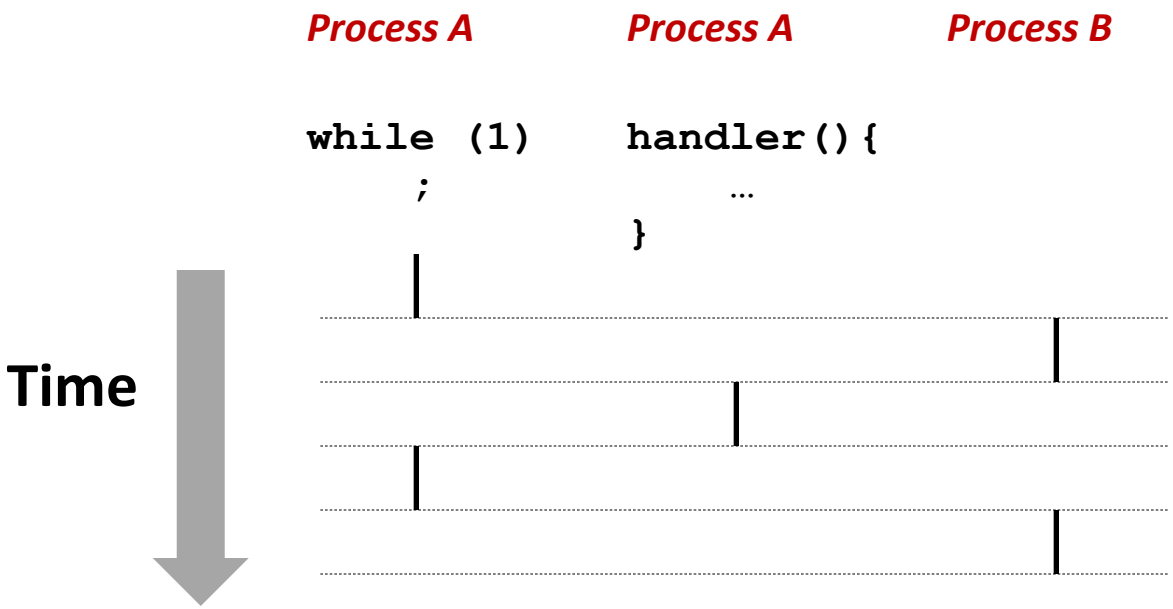
Signal Handling Example

```c
void sigint_handler(int sig) /* SIGINT handler */
{
    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

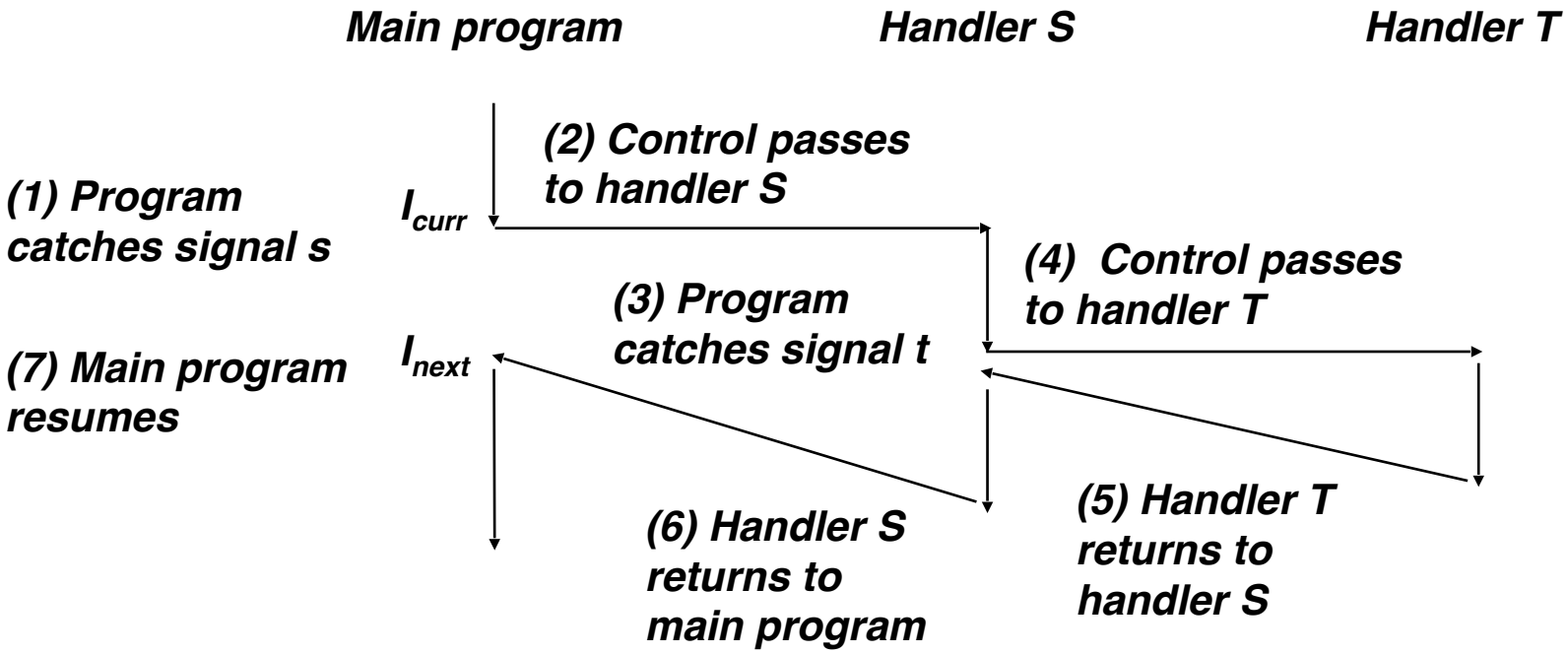
Signals Handlers as Concurrent Flows

- A signal handler is a separate logical flow (not process) that runs concurrently with the main program



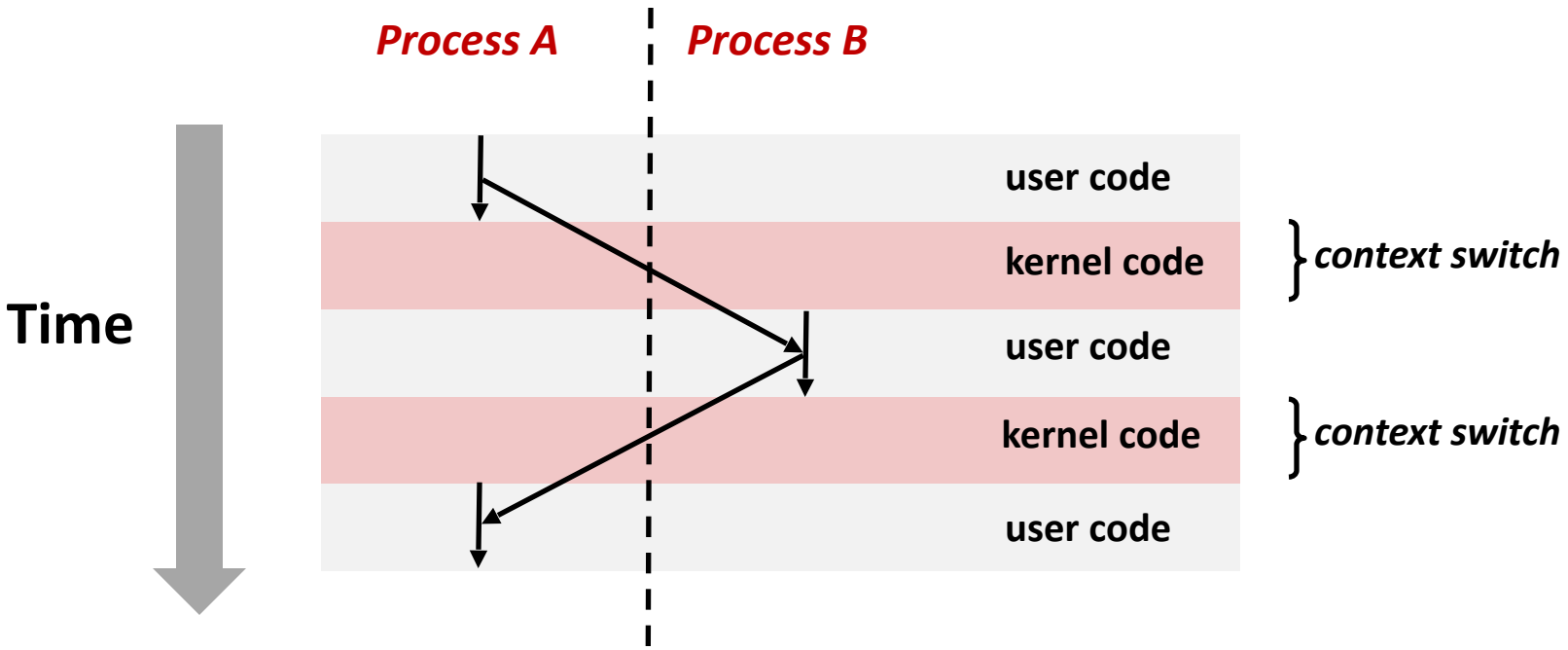
Nested Signal Handlers

- Handlers can be interrupted by other handlers



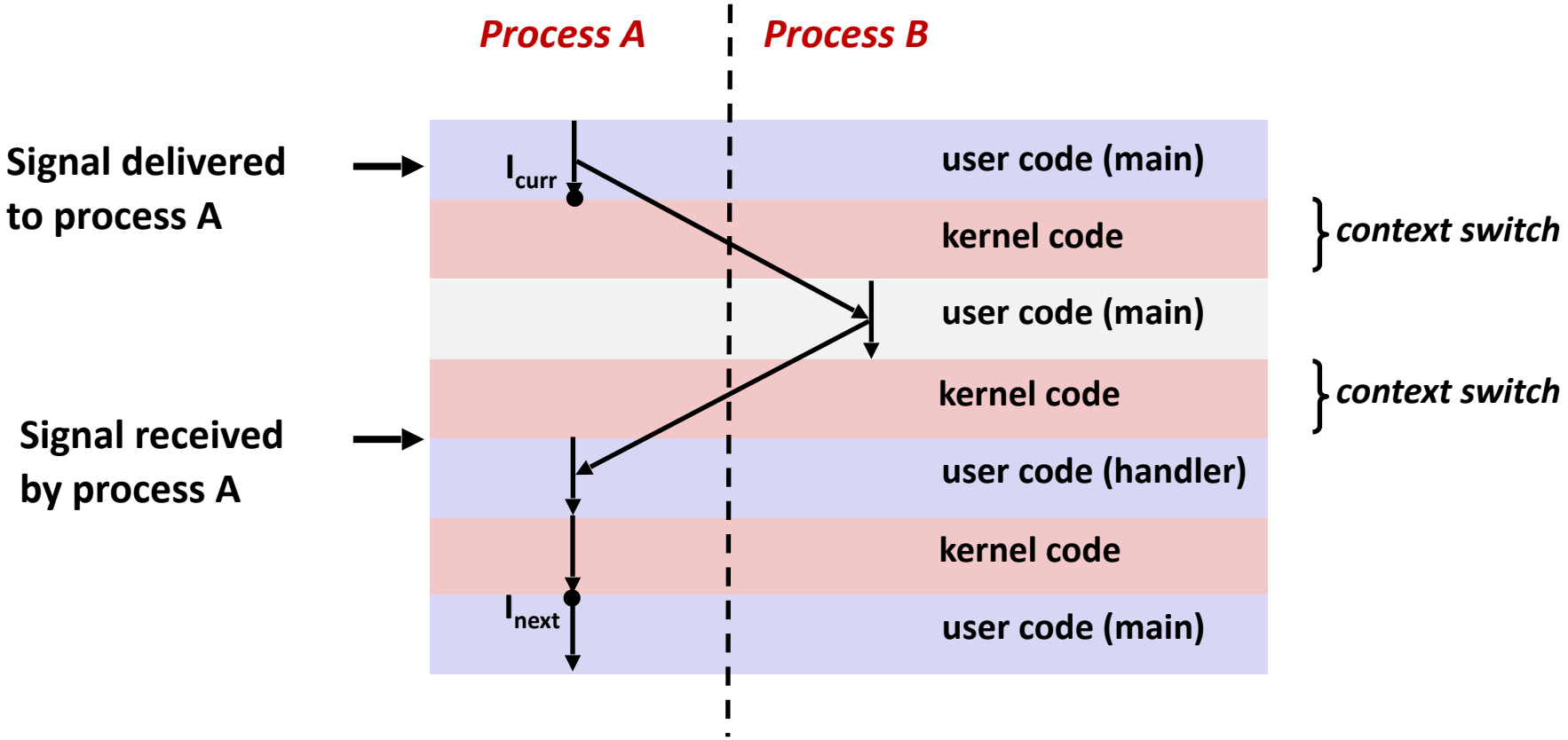
Receiving/Responding to Signals

- Kernel handles signals delivered to a process *p* **when it switches to *p* from kernel mode to user mode** (e.g., after a context switch)



Receiving/Responding to Signals

- Kernel handles signals delivered to a process *p* **when it switches to *p* from kernel mode to user mode** (e.g., after a context switch)



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 for a process
  - That is: *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 pending signal is received at most once

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 for a process
  - That is: *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 pending signal is received at most once
- A process can block/mask the receipt of certain signals

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 for a process
  - That is: *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 pending signal is received at most once
- A process can block/mask the receipt of certain signals
  - Blocked signals can be delivered, i.e., in the pending state, but will not be received/responded to *until the signal is unblocked*

Pending/Blocked Bits

- Kernel maintains pending and masked 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
  - masked: represents the set of blocked signals
    - Can be set and cleared by using the `sigprocmask` function
    - Also referred to as the signal mask.

Receiving Signals

- Right before kernel is ready to pass control to process p

Receiving Signals

- Right before kernel is ready to pass control to process p
- Kernel computes the set of pending & nonmasked signals for process p (PNM set)

Receiving Signals

- Right before kernel is ready to pass control to process p
- Kernel computes the set of pending & nonmasked signals for process p (PNM set)
- If (PNM is empty), i.e., no signal is pending & nonmasked

Receiving Signals

- Right before kernel is ready to pass control to process p
- Kernel computes the set of pending & nonmasked signals for process p (PNM set)
- If (PNM is empty), i.e., no signal is pending & nonmasked
  - No signals to respond to; simply pass control to next instruction in the logical flow for p

Receiving Signals

- Right before kernel is ready to pass control to process p
- Kernel computes the set of pending & nonmasked signals for process p (PNM set)
- If (PNM is empty), i.e., no signal is pending & nonmasked
  - No signals to respond to; simply pass control to next instruction in the logical flow for p
- Else

Receiving Signals

- Right before kernel is ready to pass control to process p
- Kernel computes the set of pending & nonmasked signals for process p (PNM set)
- If (PNM is empty), i.e., no signal is pending & nonmasked
  - No signals to respond to; simply pass control to next instruction in the logical flow for p
- Else
  - Choose least nonzero bit k in pnm and force process p to receive signal k, i.e., by executing the corresponding signal handler

Receiving Signals

- Right before kernel is ready to pass control to process p
- Kernel computes the set of pending & nonmasked signals for process p (PNM set)
- If (PNM is empty), i.e., no signal is pending & nonmasked
  - No signals to respond to; simply pass control to next instruction in the logical flow for p
- Else
  - Choose least nonzero bit k in pnm and force process p to receive signal k, i.e., by executing the corresponding signal handler
  - Repeat for all nonzero k in pnm

Receiving Signals

- Right before kernel is ready to pass control to process p
- Kernel computes the set of pending & nonmasked signals for process p (PNM set)
- If (PNM is empty), i.e., no signal is pending & nonmasked
  - No signals to respond to; simply pass control to next instruction in the logical flow for p
- Else
  - Choose least nonzero bit k in pnm and force process p to receive signal k, i.e., by executing the corresponding signal handler
  - Repeat for all nonzero k in pnm
  - Pass control to next instruction in logical flow for p

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);
```

- Explicit blocking and unblocking signal
  - `sigprocmask` function
  - `sigemptyset` – Create empty set
  - `sigfillset` – Add every signal number to set
  - `sigaddset` – Add signal number to set
  - `sigdelset` – Delete signal number from set

Safe Signal Handling

- Handlers are tricky because they are concurrent with main program and may share the same global data structures.

Safe Signal Handling

- Handlers are tricky because they are concurrent with main program and may share the same global data structures.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid, y = 0;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

Safe Signal Handling

- Handlers are tricky because they are concurrent with main program and may share the same global data structures.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid, y = 0;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

What if the following happens:

Safe Signal Handling

- Handlers are tricky because they are concurrent with main program and may share the same global data structures.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid, y = 0;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

What if the following happens:

- Parent process executes and finishes `if (x == 5)`

Safe Signal Handling

- Handlers are tricky because they are concurrent with main program and may share the same global data structures.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid, y = 0;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

What if the following happens:

- Parent process executes and finishes `if (x == 5)`
- Context switch to child, which then terminates, sends a SIGCHLD signal

Safe Signal Handling

- Handlers are tricky because they are concurrent with main program and may share the same global data structures.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid, y = 0;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

What if the following happens:

- Parent process executes and finishes `if (x == 5)`
- Context switch to child, which then terminates, sends a SIGCHLD signal
- Another context switch back to parent, and now the kernel needs to execute the SIGCHLD handler

Safe Signal Handling

- Handlers are tricky because they are concurrent with main program and may share the same global data structures.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid, y = 0;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

What if the following happens:

- Parent process executes and finishes `if (x == 5)`
- Context switch to child, which then terminates, sends a SIGCHLD signal
- Another context switch back to parent, and now the kernel needs to execute the SIGCHLD handler
- When return to parent process, **y == 20**!

Safe Signal Handling

- Handlers are tricky because they are concurrent with main program and may share the same global data structures.
  - Programmers have no control over the execution ordering between the main program and the signal handler, that is:
    - when a signal happens/delivers (depends on user or other process)
    - when the signal handler will be executed (depends on kernel)
  - If not careful, shared data structures can be corrupted

Fixing the Signal Handling Bug

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    sigfillset(&mask_all);
    signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
    if (x == 5)
        y = x * 2; // You'd expect y == 10
    Sigprocmask(SIG_SETMASK, &prev_all, NULL);

    exit(0);
}
```

- Block all signals before accessing a shared, global data structure.

Another Unsafe Signal Handler Example

Another Unsafe Signal Handler Example

- Assume a program wants to do the following:
  - The parent creates multiple child processes
  - When each child process is created, add the child PID to a queue
  - When a child process terminates, the parent process removes the child PID from the queue

Another Unsafe Signal Handler Example

- Assume a program wants to do the following:
  - The parent creates multiple child processes
  - When each child process is created, add the child PID to a queue
  - When a child process terminates, the parent process removes the child PID from the queue
- One possible implementation:
  - An array for keeping the child PIDs
  - Use a loop to fork child, and add PID to the array after fork
  - Install a handler for SIGCHLD in parent process
  - The SIGCHLD handler removes the child PID

First Attempt

```
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        /* Delete the child from the job list */
        deletejob(pid);
    }
}

int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

First Attempt

```
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        /* Delete the child from the job list */
        deletejob(pid);
    }
}

int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

The following can happen:

First Attempt

```
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        /* Delete the child from the job list */
        deletejob(pid);
    }
}

int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

The following can happen:

- The first child runs, and terminates

First Attempt

```
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        /* Delete the child from the job list */
        deletejob(pid);
    }
}

int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

The following can happen:

- The first child runs, and terminates
- Kernel sends SIGCHLD

First Attempt

```
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        /* Delete the child from the job list */
        deletejob(pid);
    }
}

int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

The following can happen:

- The first child runs, and terminates
- Kernel sends SIGCHLD
- Context switch to parent, which executes the SIGCHLD handler before **`addjob(pid)`** is executed

First Attempt

```
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        /* Delete the child from the job list */
        deletejob(pid);
    }
}

int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

The following can happen:

- The first child runs, and terminates
- Kernel sends SIGCHLD
- Context switch to parent, which executes the SIGCHLD handler before **`addjob(pid)`** is executed
- The handler deletes the job, which isn't in the queue yet!

First Attempt

```
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        /* Delete the child from the job list */
        deletejob(pid);
    }
}

int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

The following can happen:

- The first child runs, and terminates
- Kernel sends SIGCHLD
- Context switch to parent, which executes the SIGCHLD handler before **`addjob(pid)`** is executed
- The handler deletes the job, which isn't in the queue yet!
- The parent process resumes and adds a terminated child to job list

First Attempt

```
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        /* Delete the child from the job list */
        deletejob(pid);
    }
}

int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

Key in this example: creating a child and adding its PID to the job list must be an *atomic unit:* either both happen or neither happen; there can't be anything else that separates the two.

Second Attempt

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

    sigfillset(&mask_all);
    while ((pid = wait(NULL)) > 0) {
        sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid);
        sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
}
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;

    sigfillset(&mask_all);
    signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) {
            Execve("/bin/date", argv, NULL);
        }
        sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        addjob(pid);
        sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    exit(0);
}
```

Third Attempt (The Correct One)

```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, mask_one, prev_one;

    Sigfillset(&mask_all);
    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
        if ((pid = Fork()) == 0) { /* Child process */
            Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
            Execve("/bin/date", argv, NULL);
        }
        addjob(pid);  /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_one, NULL);  /* Unblock SIGCHLD */
    }
    exit(0);
}
```

Third Attempt (The Correct One)

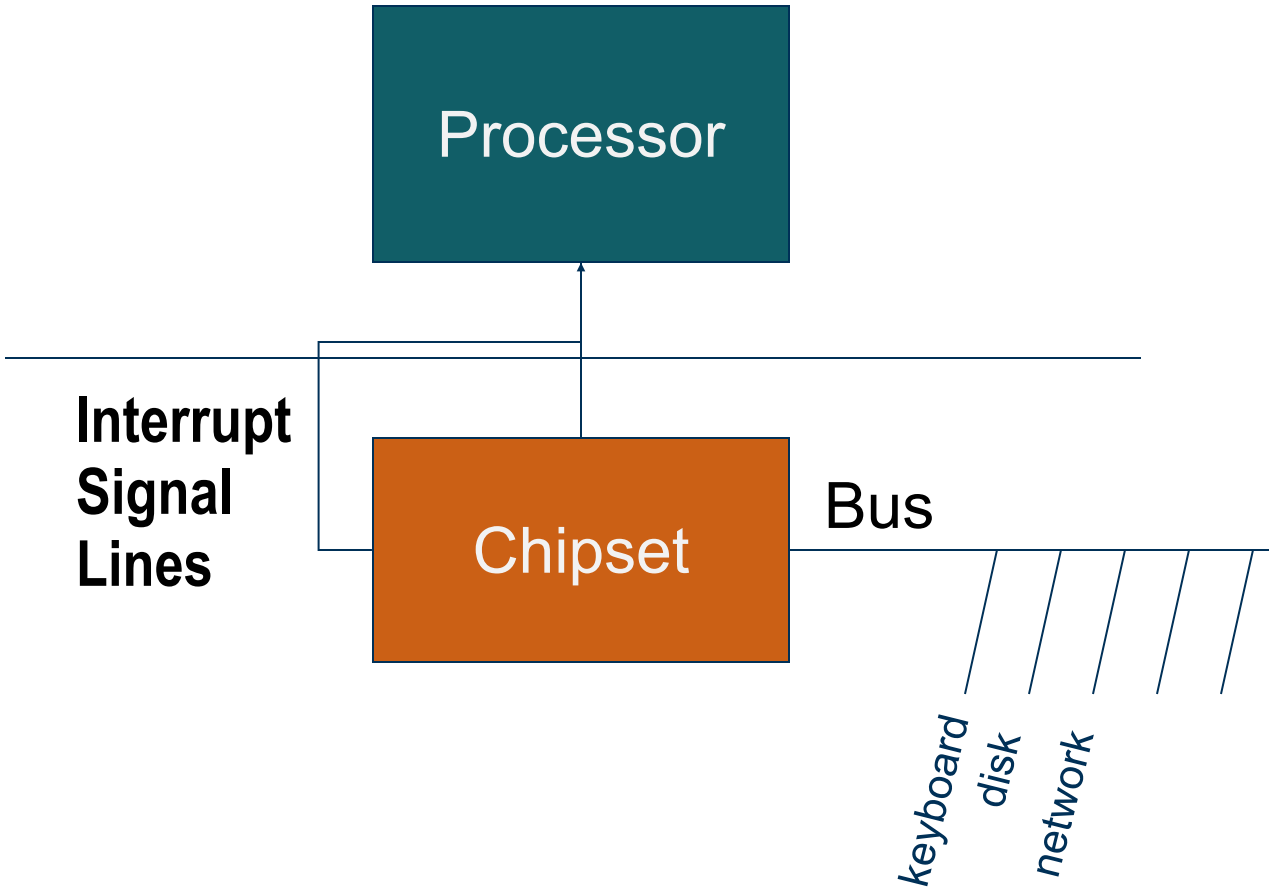
```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, mask_one, prev_one;

    Sigfillset(&mask_all);
    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
        if ((pid = Fork()) == 0) { /* Child process */
            Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
            Execve("/bin/date", argv, NULL);
        }
        addjob(pid);  /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_one, NULL);  /* Unblock SIGCHLD */
    }
    exit(0);
}
```

Why this? →

Today

- Signals: The Way to Communicate with Processes
- Interrupts and exceptions: how signals are triggered

Interrupts in a Processor



Interrupts, a.k.a., Asynchronous Exceptions

- Caused by events external to the processor
  - Events that can happen at any time. Computers have little control.
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction

Interrupts, a.k.a., Asynchronous Exceptions

- Caused by events external to the processor
  - Events that can happen at any time. Computers have little control.
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction

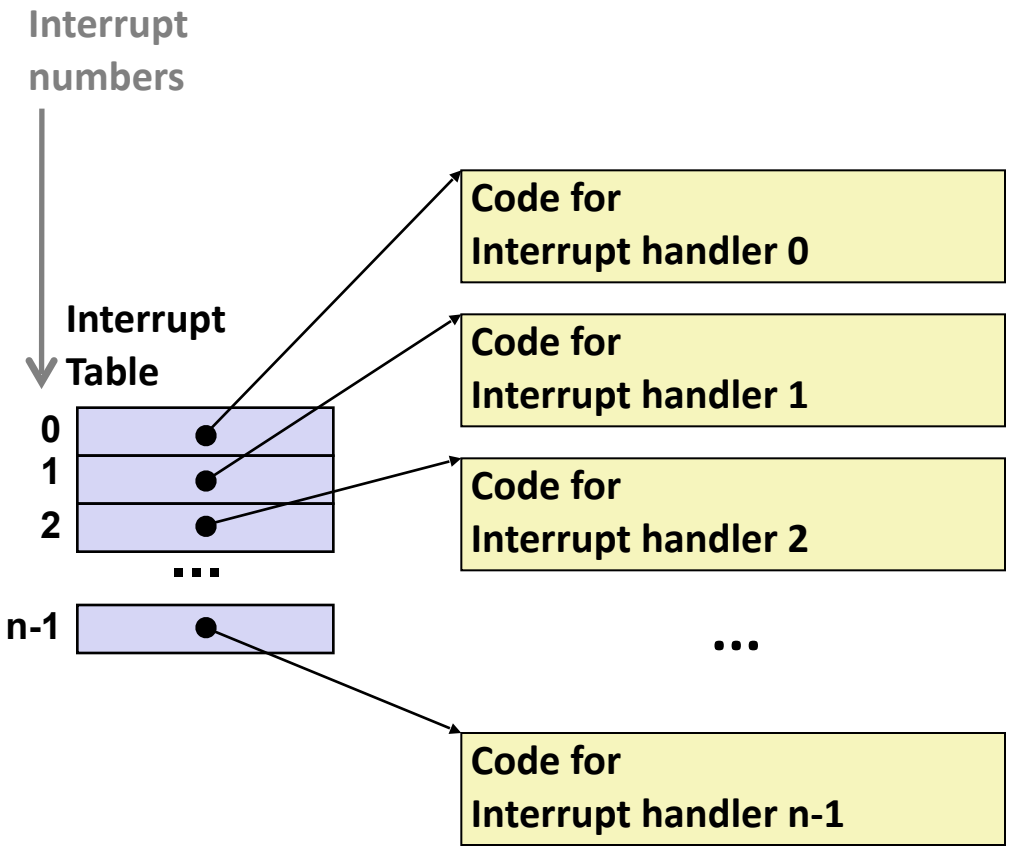
- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt

Interrupts, a.k.a., Asynchronous Exceptions

- Caused by events external to the processor
  - Events that can happen at any time. Computers have little control.
  - 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

Each Interrupt Has a Handler



- Each type of event has a unique interrupt number k
- k = index into interrupt table
- Interrupt table lives in memory. Its start address is stored in a special register
- Handler k is called each time interrupt k occurs

Sending Signals from the Keyboard

- Can you guess how Ctrl + C might be implemented?

Sending Signals from the Keyboard

- Can you guess how Ctrl + C might be implemented?
  - Ctrl + C sends a keyboard interrupt to the CPU, which triggers an interrupt handler

Sending Signals from the Keyboard

- Can you guess how Ctrl + C might be implemented?
  - Ctrl + C sends a keyboard interrupt to the CPU, which triggers an interrupt handler
  - The interrupt handler, executed by the kernel, triggers certain piece of the kernel, which generates the signal, which is then delivered to the target process

When to Execute the Handler?

- Interrupts: when convenient. Typically wait until the current instructions in the pipeline are finished
- Maskable verses Unmaskable
  - Interrupts can be individually masked (i.e., ignored by CPU)
  - Synchronous exceptions are usually unmaskable
- Some interrupts are intentionally unmaskable
  - Called non-maskable interrupts (NMI)
  - Indicating a critical error has occurred, and that the system is probably about to crash

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - *Traps*
    - Intentional
    - Examples: *system calls*, breakpoint traps, special instructions
  - *Faults*

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - *Traps*
    - Intentional
    - Examples: *system calls*, breakpoint traps, special instructions
  - *Faults*
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), ***protection faults (the infamous Segmentation Fault!)*** (unrecoverable in Linux), floating point exceptions (unrecoverable in Linux)
    - These exceptions will generate signals to processes
  - *Aborts*

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - *Traps*
    - Intentional
    - Examples: *system calls*, breakpoint traps, special instructions
  - *Faults*
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), ***protection faults (the infamous Segmentation Fault!)*** (unrecoverable in Linux), floating point exceptions (unrecoverable in Linux)
    - These exceptions will generate signals to processes
  - *Aborts*
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error
    - Aborts current program through a SIGABRT signal

Where Do You Restart?

- Interrupts/Traps
  - Handler returns to the ***following*** instruction

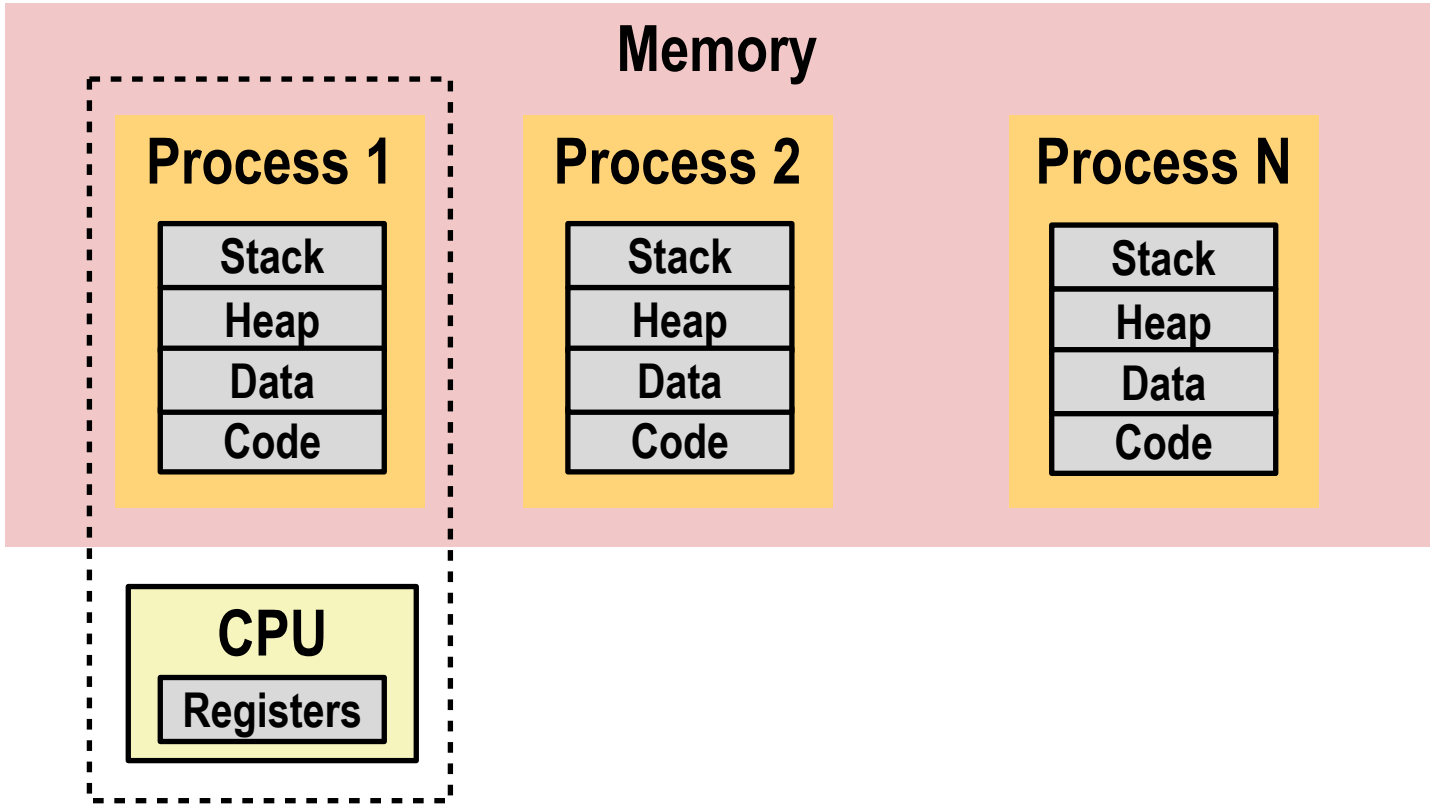
Where Do You Restart?

- Interrupts/Traps
  - Handler returns to the ***following*** instruction
- Faults
  - Exception handler returns to the instruction that caused the exception, i.e., ***re-execute*** it!

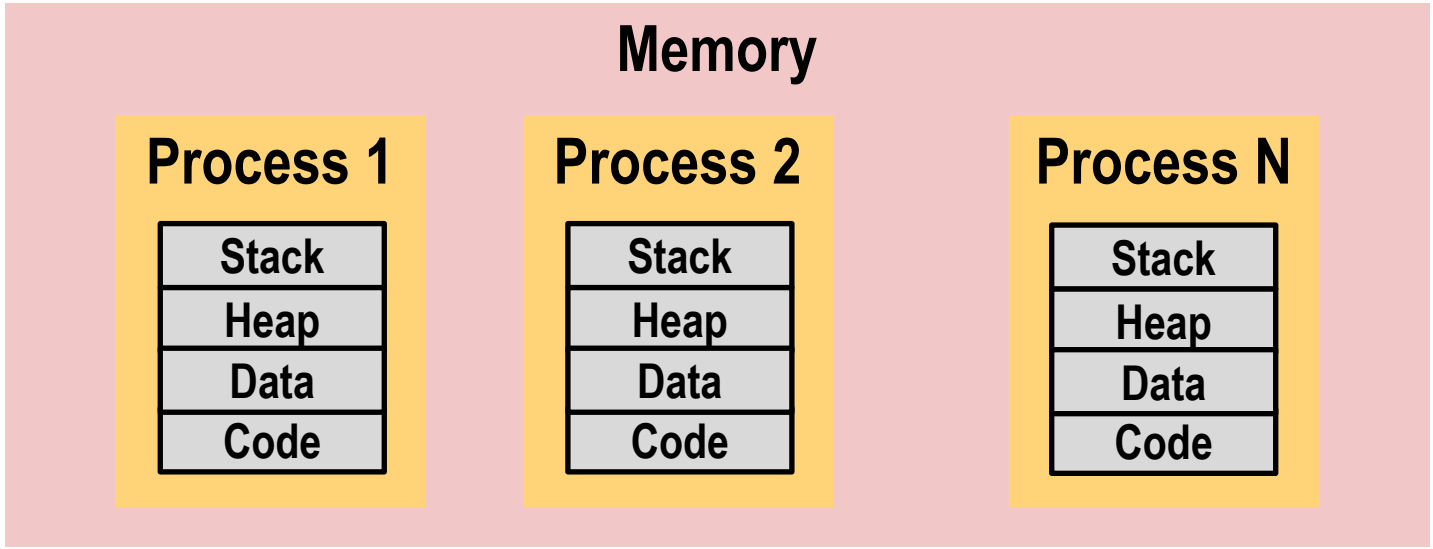
Where Do You Restart?

- Interrupts/Traps
  - Handler returns to the ***following*** instruction
- Faults
  - Exception handler returns to the instruction that caused the exception, i.e., ***re-execute*** it!
- Aborts
  - Never returns to the program

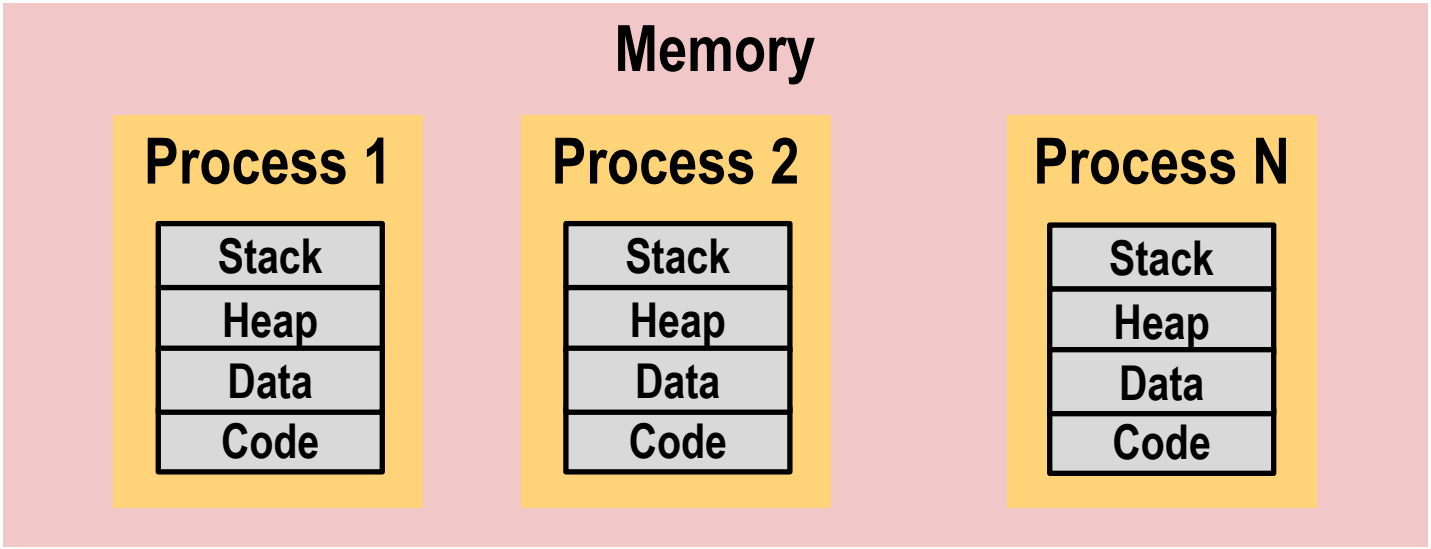
Multiprocessing Illustration



Problem 1: Space



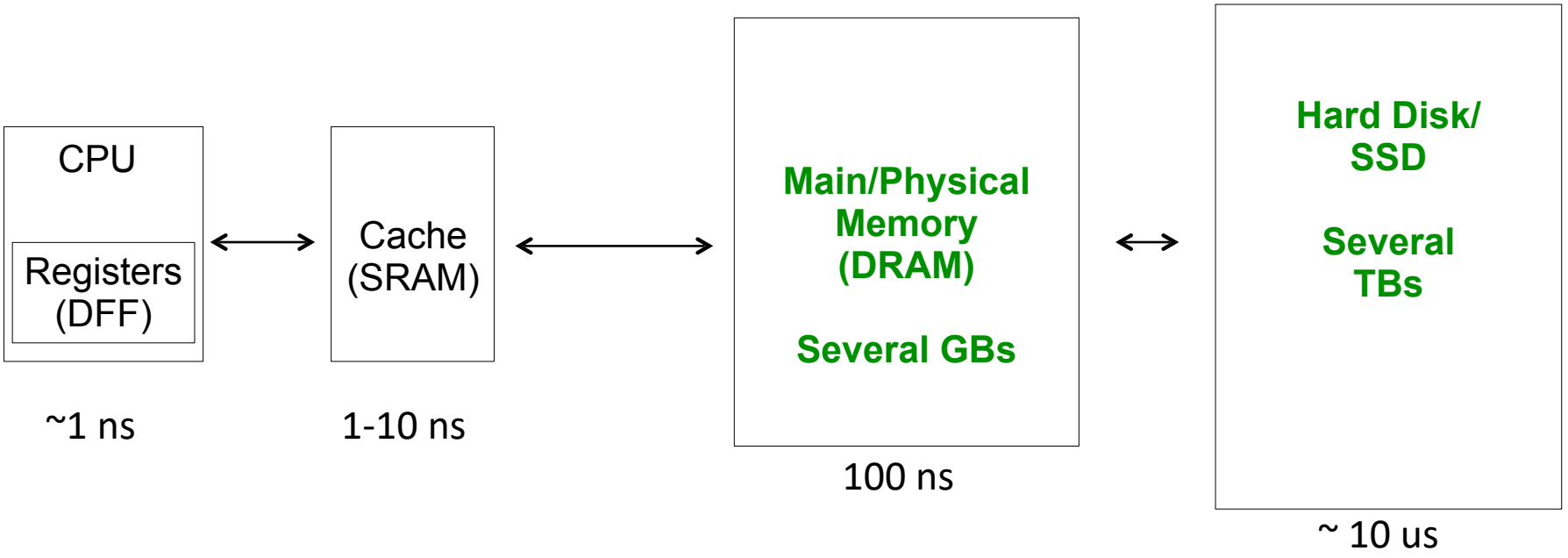
Problem 1: Space



- Space:
  - Each process's address space is huge (64-bit): can memory hold it (16GB is just 34-bit)?
  - 2^48 bytes is 256 TB
  - There are multiple processes, increasing the storage requirement further

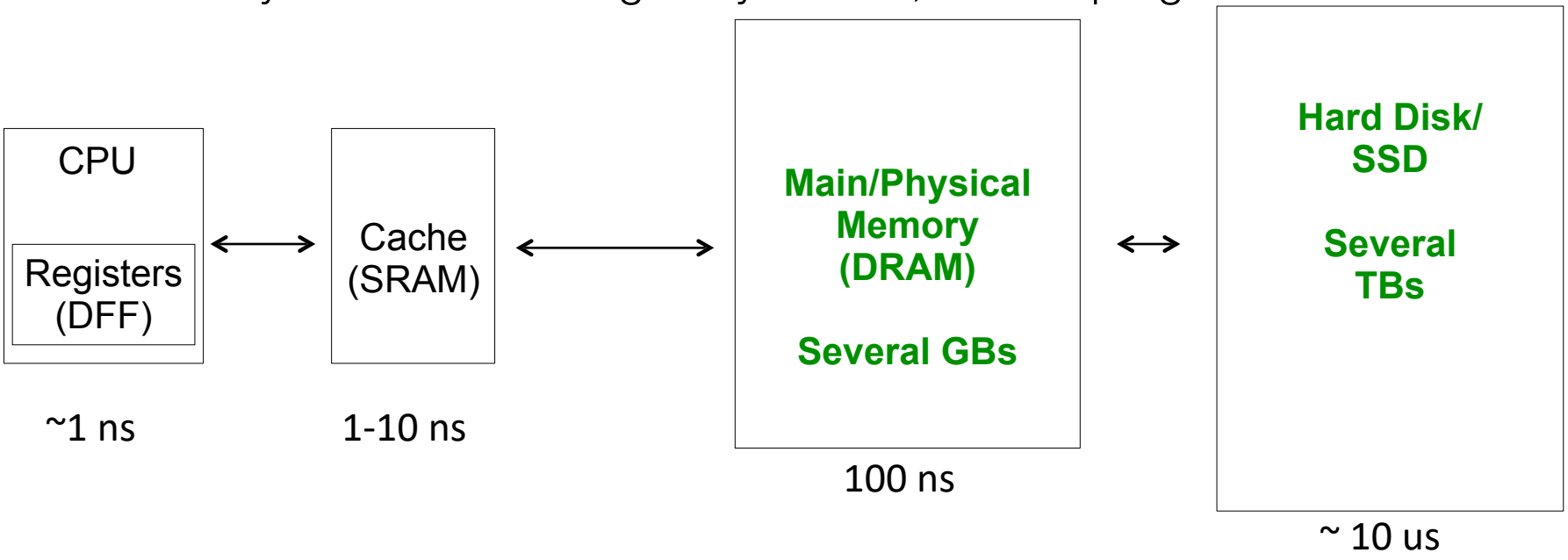
Recall: Memory Hierarchy

- Solution: store all the data in disk (several TBs typically), and use memory only for most recently used data
  - Of course if a process uses all its address space that won't be enough, but usually a process won't use all 64 bits. So it's OK.



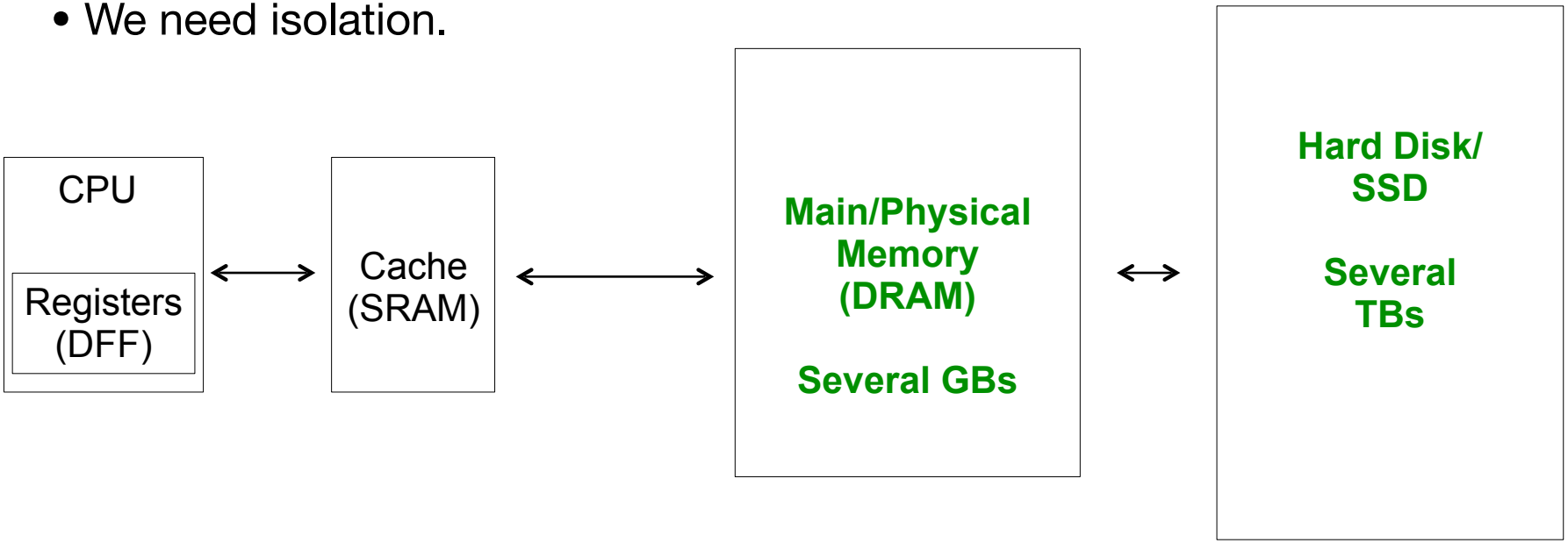
Recall: Memory Hierarchy

- Solution: store all the data in disk (several TBs typically), and use memory only for most recently used data
  - Of course if a process uses all its address space that won't be enough, but usually a process won't use all 64 bits. So it's OK.
- Challange: who is moving data back and forth between the DRAM/main memory/physical memory and the disk?
  - Ideally should be managed by the OS, not the programmer.



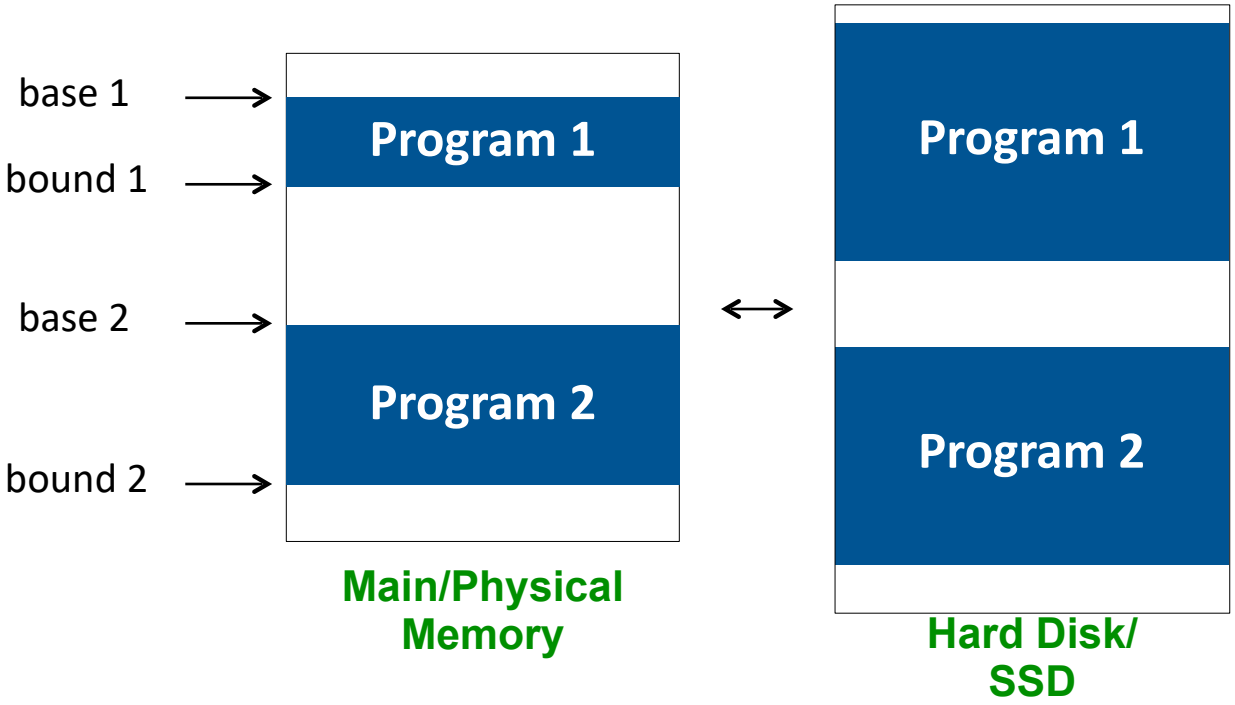
Problem 2: Security

- Different programs/processes will share the same physical memory
  - Or even different uses. A CSUG machine is accessed by all students, but there is one single physical memory!
- What if a malicious program steals/modifies data from your program?
  - If the malicious program get the address of the memory that stores your password, should it be able to access it? If not, how to prevent it?
- We need isolation.



One Way to Isolate: Segments

- Different processes will have exclusive access to just one part of the physical memory.
- This is called ***Segments***.
  - Need a base register and a bound register for each process. Not widely used today. x86 still supports it (backward compatibility!)
  - Fast but not inflexible. Makes benign sharing hard.



Problem 3: Fragmentation (with Segments)

- Each process gets a continuous chunk of memory. Inflexible.
- What if a process requests more space than any continuous chunk in memory but smaller than the total free memory?
  - This is called “fragmentation”; will talk about this more later.
- Need to allow assigning discontinuous chunks of memory to processes.