

CSCI 3753

Operating Systems

Interprocess Communication

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 02/06/13

Communicating Between Processes

- **Inter-Process Communication (IPC):** Want to communicate between two processes because (why?)
 - An application may split its tasks into two or more processes for reasons of convenience and/or performance
 - e.g. Web server and cgi-bin (Common Gateway Interface) – for certain URLs, Web server creates a new cgi-bin process to service the request, e.g. dynamic content generation
 - Creating a new process is expensive
 - Sharing information
 - Improved fault isolation

Communicating Between Processes

- Two types of IPC

1. *shared memory* - OS provides mechanisms that allow creation of a shared memory buffer between processes

- `shmid = shmget (key name, size, flags)` is part of the POSIX API that creates a shared memory segment, using a name (key ID)
 - All processes sharing the memory need to agree on the key name in advance.
 - Creates a new shared memory segment if no such shared memory with the same name exists and returns handle to the shared memory. If it already exists, then just return the handle.

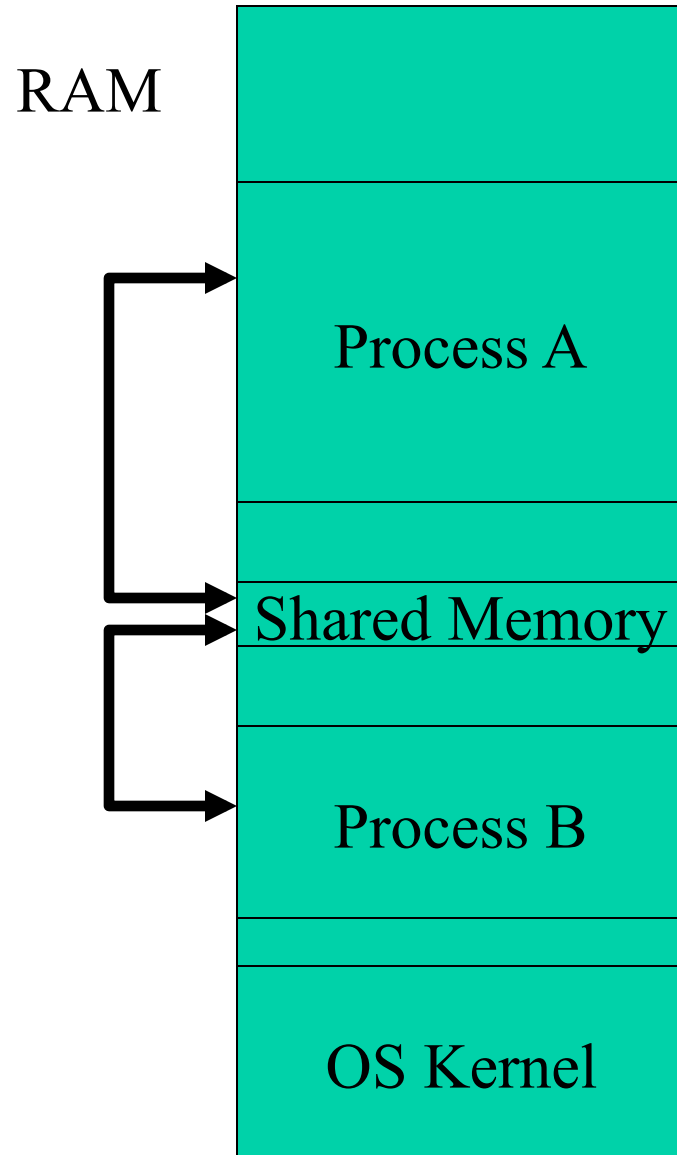
Communicating Between Processes (2)

- Two types of IPC

- 1. *shared memory* (cont.)

- `shm_ptr = shmat (shmid, NULL, 0)` to attach a shared memory segment to a process' s address space
 - This association is also called binding
 - Reads and writes now just use `shm_ptr`
 - `shmctl()` to modify control information and permissions related to a shared memory segment, & to remove a shared memory segment

IPC Shared Memory

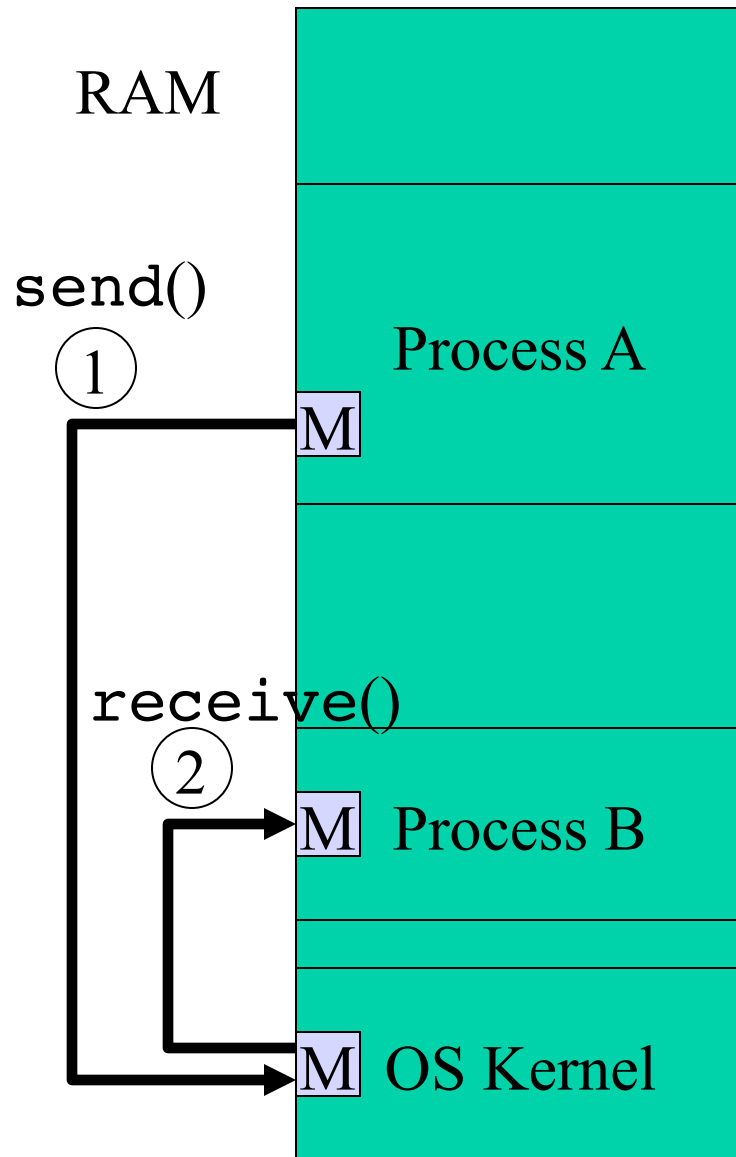


- allows fast and high volume reading/writing of buffer in memory
- applies to processes on the same machine
- Problem: shared access to the same memory introduces complexity
 - need to synchronize access
 - Producer-Consumer example
 - if two producers write at the same time to shared memory, then they can overwrite each other's data
 - if a producer writes while a consumer is reading, then the consumer may read inconsistent data

IPC Message Passing

- Two types of IPC
 2. *message passing* - OS provides constructs that allow communication via buffers
 - Basic primitives are `send()` and `receive()`
 - typically implemented via system calls, and is hence slower than shared memory
 - Sending process has a buffer to send/receive messages, as does the receiving process and OS
 - In direct message-passing, processes send directly to each other's buffers
 - In indirect message-passing, sending process sends a message to OS by calling `send()`. OS acts as a buffer or mailbox for relaying the message to the receiving process, which calls `receive()` to retrieve message

IPC Message Passing



Example of indirect message-passing

- `send()` and `receive()` can be blocking/synchronous or non-blocking/asynchronous
- used to pass small messages
- Advantage: doesn't require synchronization
- Disadvantage: Slow - OS is involved in each IPC operation for control signaling and possibly data as well
- Message Passing IPC types: pipes, UNIX-domain sockets, Internet domain sockets, message queues, and remote procedure calls (RPC)

IPC via Pipes

- Process 1 writes into one end of the pipe, & process 2 reads from other end of the pipe, e.g. “ls | more”
 - Form of IPC similar to message-passing but data is viewed as a stream of bytes rather than discrete messages
 - was one of UNIX’ s original forms of IPC
 - essentially FIFO buffers accessed like file I/O API, so standard read() and write() for files can be used
 - Asynchronous/non-blocking send() and blocking/synchronous receive()
- This is a one-way pipe
- This also called an *anonymous* pipe in Windows
 - Parent process uses pipe() system call to create pipe

IPC via Pipes (2)

Unix-style pseudocode example:

```
int pid;
```

```
int piped[2];
```

piped[0] is file descriptor
to read end of the pipe

piped[1] is file descriptor
to write end of pipe

```
pipe(piped);
```

```
pid = fork();
```

```
If (pid==0) { /* child */
```

```
    /* child blocks on read */
```

```
    read(piped[0], readdata, length)
```

```
} elseif (pid>0) { /* parent */
```

```
    /* parent writes data to child */
```

```
    write(piped[1], writedata, length);
```

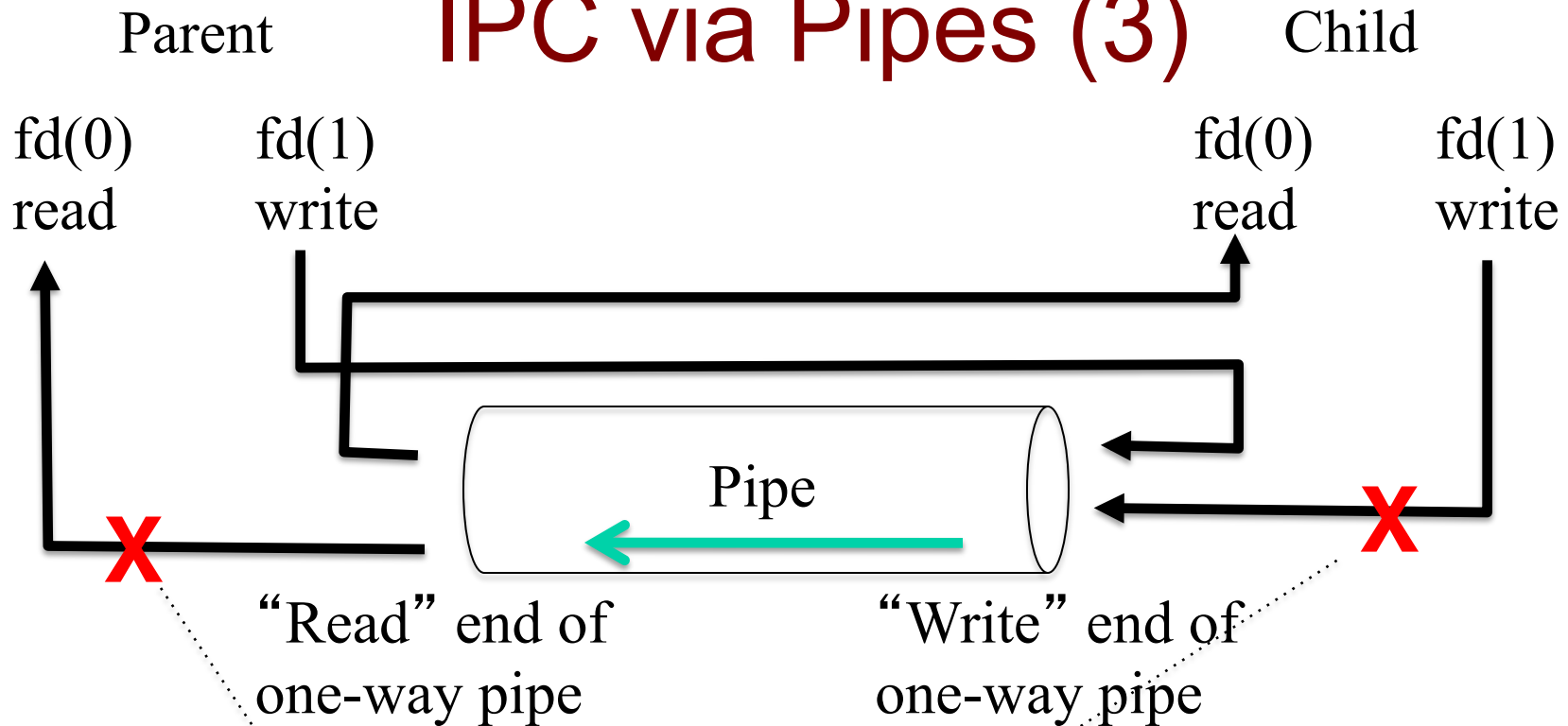
```
}
```

System call to create pipe

Once there are *length*
bytes, read returns, so as
the parent sender streams
bytes, the child reader
can process them

Send message
to child

IPC via Pipes (3)



- Chapter 3 textbook example is more detailed, e.g. closes the unused write fd for child and closes the unused read fd for parent

Named Pipes

- Traditional one-way or anonymous pipes only exist transiently between the two processes connected by the pipe
 - As soon as these processes complete, the pipe disappears
- Named pipes persist across processes
 - Operate as FIFO buffers or files, e.g. created using `mkfifo(unique_pipe_name)` on Unix
 - Different processes can attach to the named pipe to send and receive data
 - Need to explicitly remove the named pipe
 - See textbook for more info on named pipes

Using Sockets for UNIX IPC

Sockets are an example of message-passing IPC. Created in UNIX using `socket()` call:

```
sd = socket(int domain, int type, int protocol);
```

socket descriptor

- = `PF_UNIX` for local Unix domain sockets (local IPC)
- = `PF_INET` for Internet sockets (but can still achieve local communication by specifying localhost address as destination)
- = `SOCK_STREAM` for reliable in-order delivery of a byte stream
- = `SOCK_DGRAM` for delivery of discrete messages
- = 0 usually to select default protocol associated with a type

Using Sockets for UNIX IPC (2)

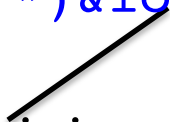
- Each communicating process will first create its own socket, usually `SOCK_STREAM`.
- Now let's consider only the case of UNIX domain sockets (`PF_UNIX` domain):
 - Used only for local communication only among a computer's processes
 - Emulates reading/writing from/to a file
 - Each process `bind()`'s its socket to a filename:

```
bind(sd, (struct sockaddr *)&local, length);
```

socket
descriptor



data structure containing unique unused file
name, e.g. `"/users/shiv/myipcsocketfile"`



Using Sockets for UNIX IPC (3)

- UNIX domain sockets (cont.):
 - Usually, one process acts as the server, and the other processes connect to it as clients

Server code:

```
sd = socket
    (PF_UNIX, SOCK_STREAM, 0)
bind(sd, ...)
```

bind and connect must
use same file name!

```
listen() for connect requests
```

```
sd2 = accept() a connect  
request
```

```
recv(sd2, ...) and  
send(sd2, ...)
```

Client code:

```
sd = socket
    (PF_UNIX, SOCK_STREAM, 0)
```

```
connect(sd, ...) to server
```

```
recv(sd, ...) and  
send(sd, ...)
```

IPC

IPC via Internet domain sockets

- Set up and connect Internet sockets in a way very similar to Unix domain sockets, but...
 - Configure the socket with domain `PF_INET` instead and set destination to localhost (say `127.0.0.1`) instead of the usual remote Internet IP address
 - And need to choose a well-known port # that is shared between processes, i.e. P1 and P2 know this port # in advance – this is similar to the well-known file name
 - Both processes then `send()` and `receive()` messages via this port and socket
 - Advantage of writing applications to use this type of IPC via Internet domain sockets is that it is arguably more portable than UNIX-domain sockets

Signals

- Are also a form of inter-process communication, but of limited control information, not data
- Used to inform processes of unexpected external events such as a time out or forced termination of a process
- Allows one process to interrupt another process and send it a coded signal using OS signaling mechanisms
 - A signal is an indication that notifies a process that some event has occurred
- Textbook only briefly explains this concept. We'll cover it in more detail.
 - It is a controlled form of IPC
 - More precisely, some signals are useful in process-to-process communication (IPC)
 - Other signals are primarily for OS-to-process communication

Signals

- Without signals, low-level hardware exceptions are processed by the kernel's exception handlers only, and are not normally visible to user processes
 - e.g. a user process would block on a system call, say `read()`, to be notified that an I/O event (completion) has occurred
 - Signals expose occurrences of such low-level exceptions to user processes
- 30 types of signals on Linux/UNIX systems
- Windows does not explicitly support signals, but does have *asynchronous procedure calls* (APCs) that emulate signals

Linux/UNIX Signals

Number	Name/Type	Event
2	SIGINT	Interrupt from keyboard (Ctrl-C)
8	SIGFPE	Floating point exception (arith. error)
9	SIGKILL	Kill a process
10, 12	SIGUSR1, SIGUSR2	User-defined signals
11	SIGSEGV	invalid memory ref (seg fault)
14	SIGALRM	Timer signal from alarm function
29	SIGIO	I/O now possible on descriptor

Sending Signals

- Kernel sends a signal to a destination process by updating some state in the context of the destination process, then waking the process up to handle the signal
- A process can send a signal to itself using a library call like `alarm()` – we'll see this example later. This still goes through the OS.
- A process X can send a signal to process Y by calling `kill(process_id, signal_num)`
 - for example, `kill(Y, SIGUSR1)` sends a `SIGUSR1` signal to process Y, which will know how to interpret this signal
 - This call still goes through the OS, not directly from process to process.

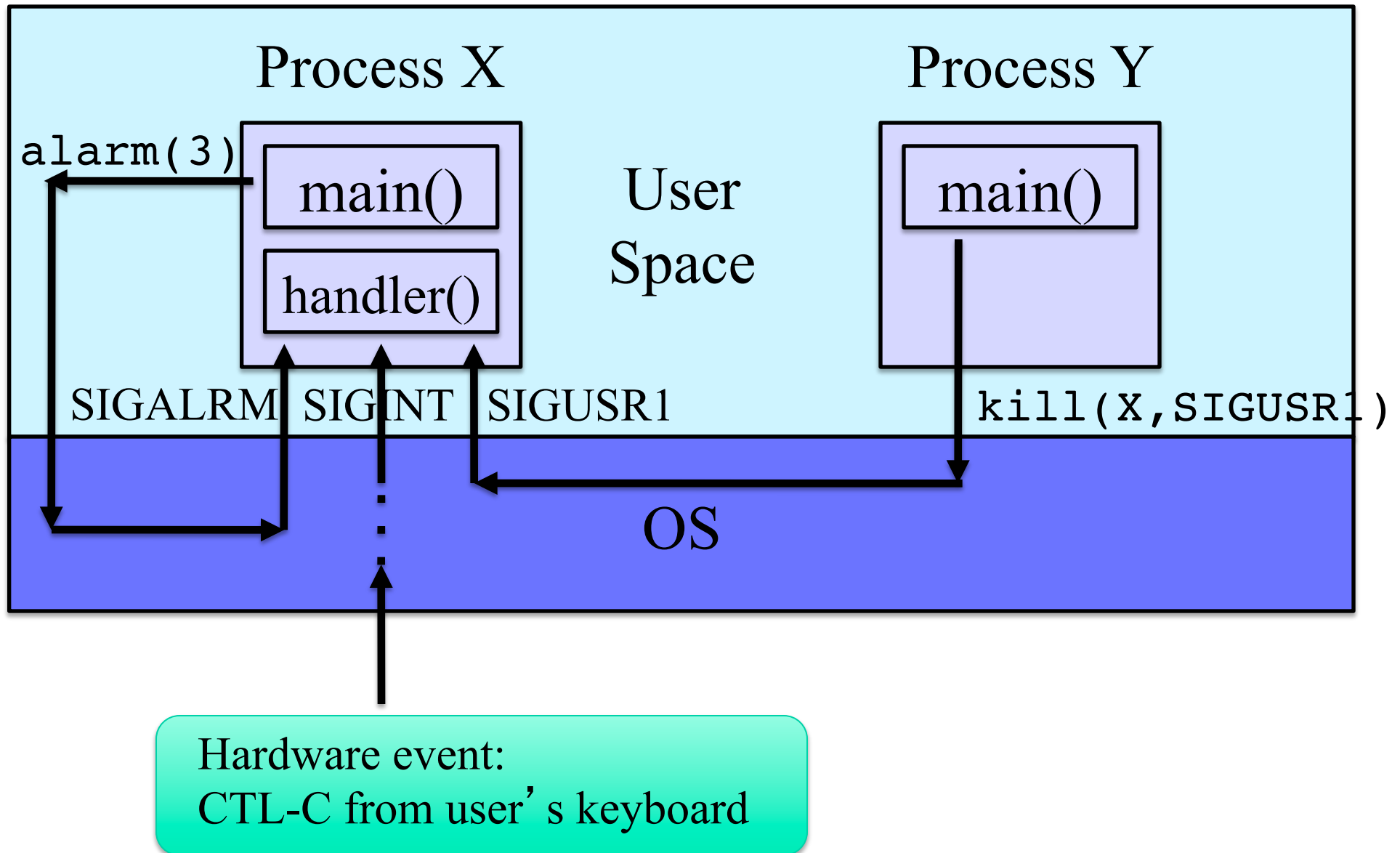
Receiving Signals

- When a kernel is returning from some exception handler, it checks to see if there are any pending signals for a process before passing control to the process
- The user may register a signal handler () via the `signal()` function. If no handler is registered, then default action is typically termination
- `signal(signum, handler)` function is used to change the action associated with a signal
 - if handler is `SIG_IGN`, then signals of type `signum` are ignored
 - if handler is `SIG_DFL`, then revert to default action, usually termination
 - otherwise if there is a user-defined function `handler`, then call it to handle the signal.

Catching Signals

- Invocation of the signal handler is called *catching the signal*. We use this term interchangeably with *handling the signal*
- The user-specified signal handler executes in the context of the affected process
 - The kernel calls the user-specified handler, and passes control to user space. When the handler is done (call returns), control returns to the kernel.
 - Note: the next time the process executes, it will resume back at the instruction where the process was originally interrupted/signaled

Signals



Signaling Example

- A process can send SIGALRM signals to itself by calling the alarm function
 - alarm(T seconds) arranges for kernel to send a SIGALRM signal to calling process in T seconds
 - see code example next slide
 - `#include<signal.h>`
 - uses `signal` function to install a signal handler function that is called asynchronously, interrupting the infinite while loop in main, whenever the process receives a SIGALRM signal
 - When handler returns, control passes back to main, which picks up where it was interrupted by the arrival of the signal, namely in its infinite loop

Signaling Example (2)

```
#include <signal.h>
int beeps=0;
```

```
void handler(int sig) {
    if (beeps<5) {
        alarm(3);
        beeps++;
    } else {
        printf("DONE\n");
        exit(0);
    }
}
```

Signal handler, passed signal #

cause next SIGALRM to be sent to this process in 3 seconds. Assume that SIGALRM is the only signal handled. Otherwise, would need a *case* statement in handler for other signals.

```
int main() {
    signal(SIGALRM, handler);
    alarm(3);

    while(1) { ; }
    exit(0);
}
```

register signal handler

cause first SIGALRM to be sent to this process in 3 seconds

infinite loop that gets interrupted by signal handling

Signals

- More generally, when a process catches a signal of type `signum=k`, the handler installed for signal `k` is invoked with a single integer argument set to `k`
- This argument allows the same handler function to catch different types of signals.

Signals

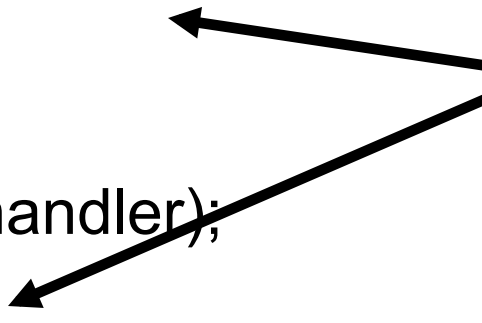
- Signals are an *asynchronous* signaling mechanism in UNIX
 - A process or thread never knows when a signal will occur. Its execution can be interrupted at any time. A process must be written to handle this asynchrony. Otherwise, could get race conditions.

Example:

```
int global=10;  
handler(int signum) {global++;}
```

```
main() {  
    signal(SIGUSR1,handler);  
    while(1) {global--;}  
}
```

- both the main program and the signal handler are manipulating the global variable – could have a *race condition* if main is interrupted in mid-execution of global-- by signal handling of SIGUSR1



Signals

- In addition, if there are multiple signals, can have a race conditions
 - i.e. if a signal handler is processing signal S1 but is interrupted by another signal S2, then could have a race condition inside the handler.
 - In the previous example, we'd have global++ happening in two handlers in rapid succession, could lead to unpredictable results.
 - The solution is to block other signals while handling the current signal.