CSC 252: Computer Organization
Spring 2025: Lecture 20

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

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);
}
```

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);
}
```

Why this? →

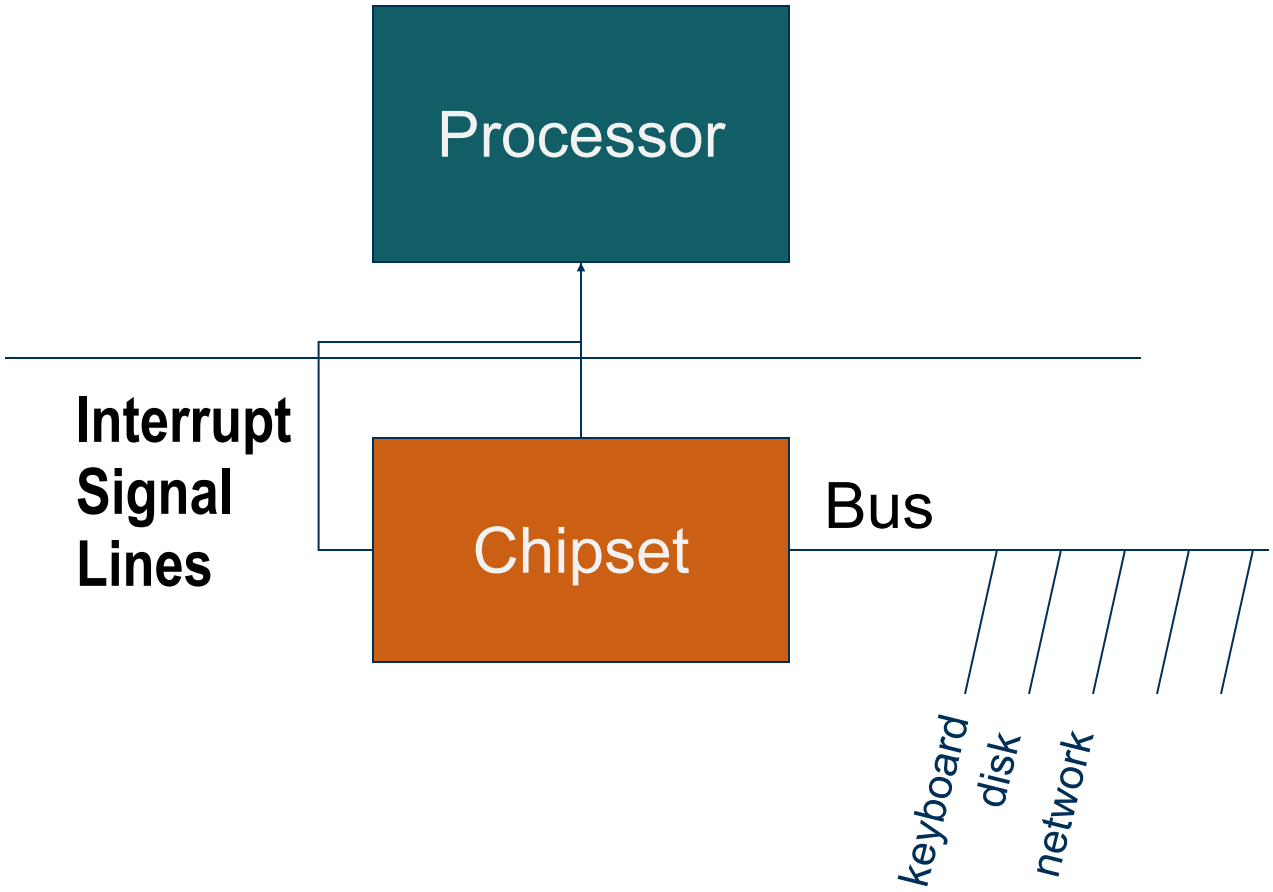
Thinking in Parallel is Hard

Maybe Thinking is Hard

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

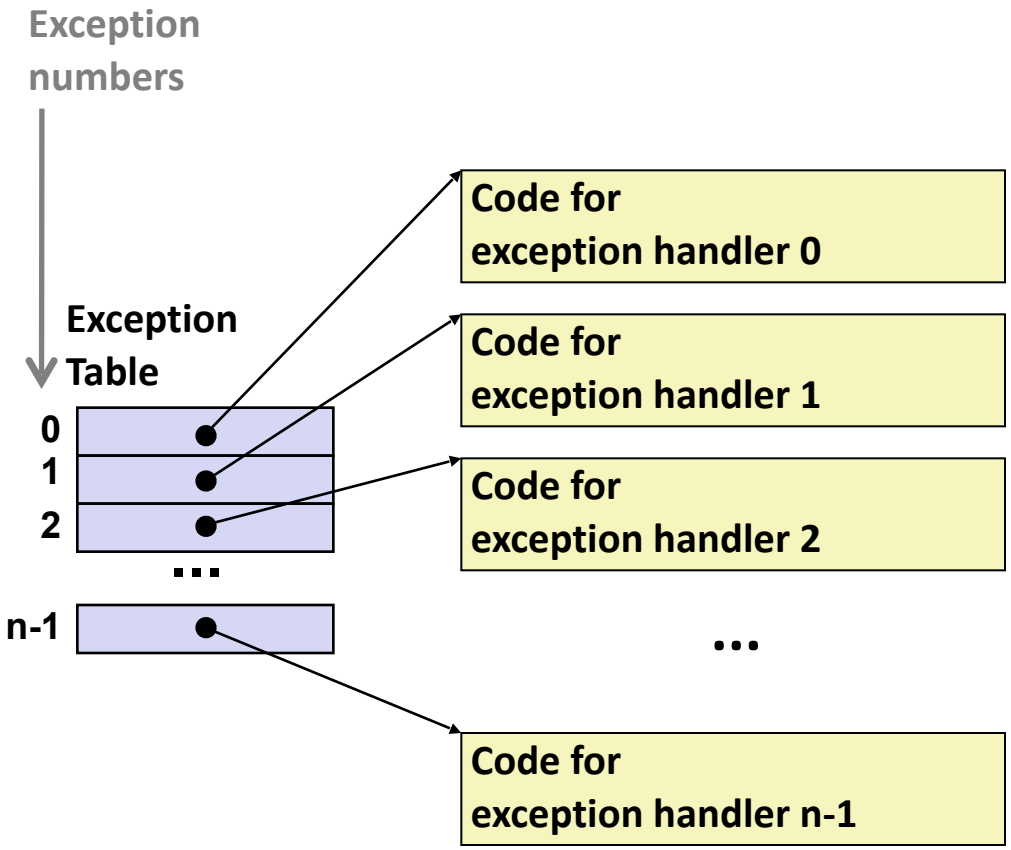
- Caused by events external to the processor
  - Events that can happen at any time. Computers have little control.
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction

- 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
    - 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, machine check
    - Aborts current program through a SIGABRT signal

Each Exception Has a Handler



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

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 SIGINT 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
- Exceptions: typically immediately as programs can't continue without resolving the exception (e.g., page fault)
- 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

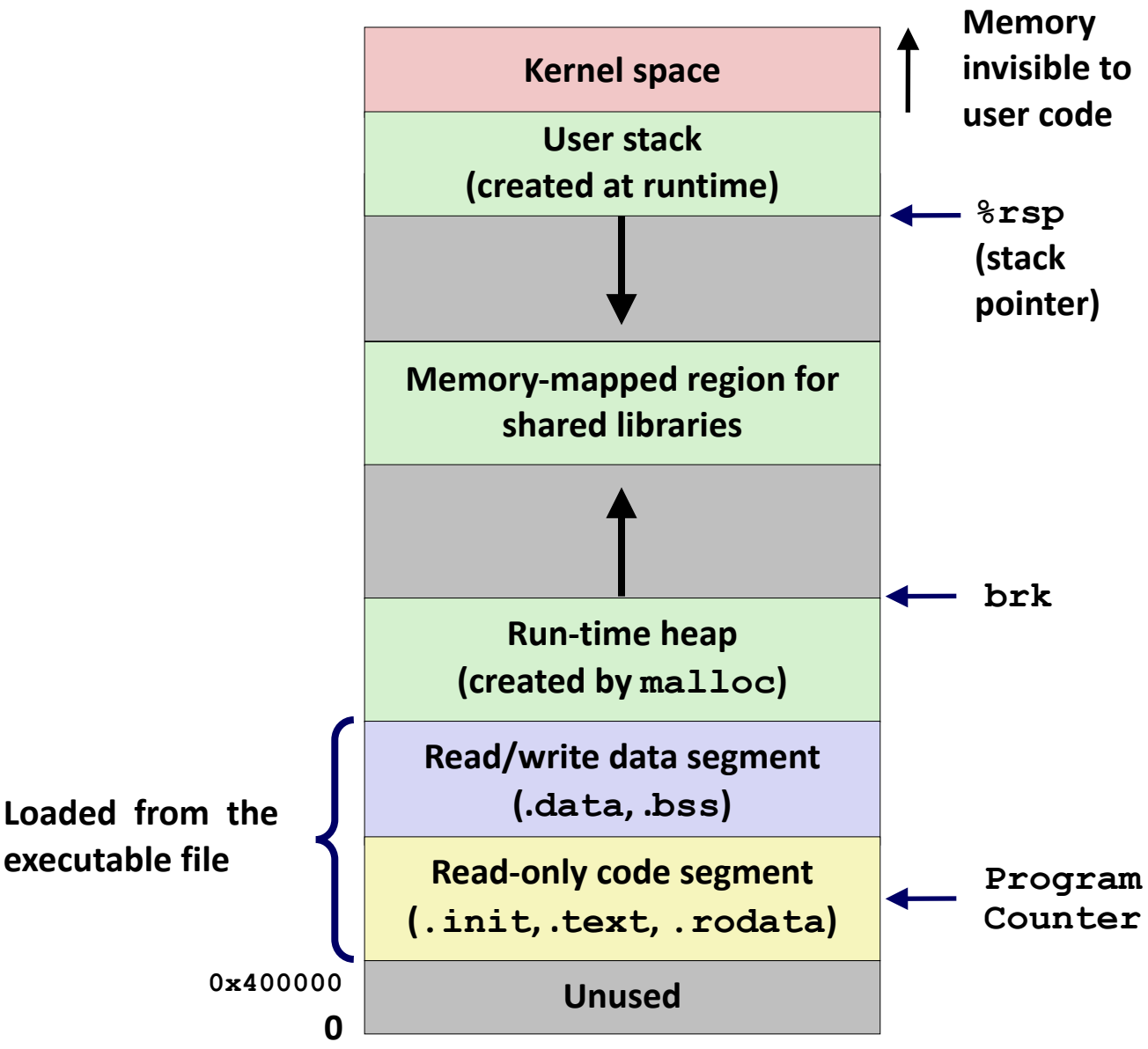
Where Do You Restart?

- 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

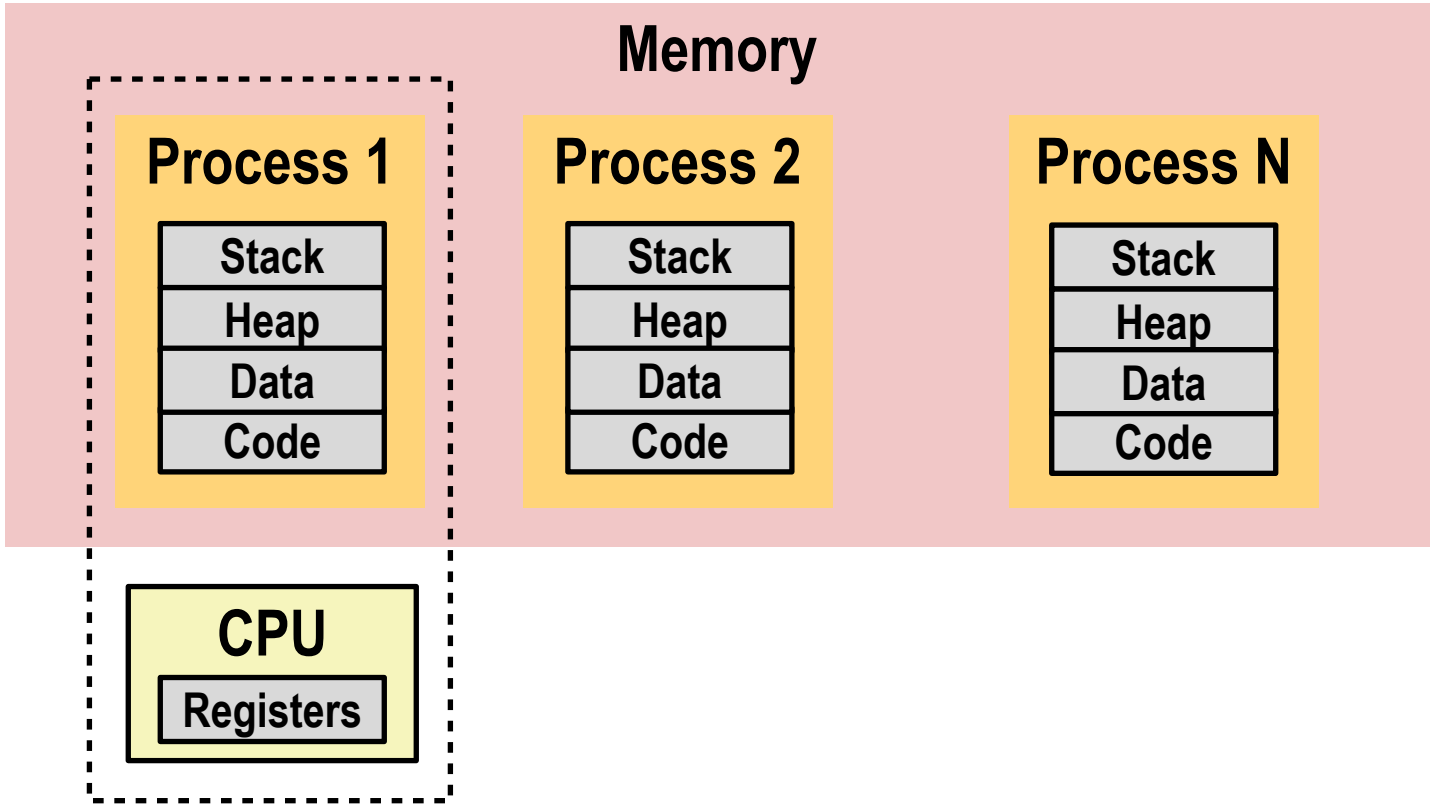
Today

- Virtual memory ideas
- VM basic concepts and operation
- Other critical benefits of VM
- Address translation

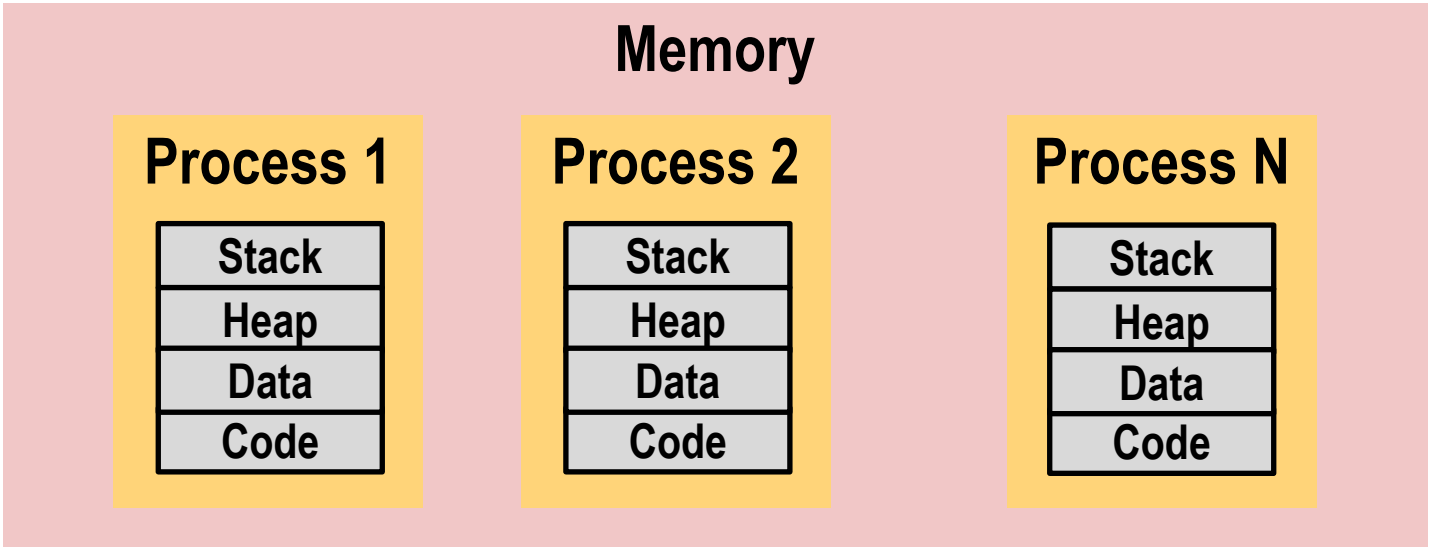
Process Address Space



Multiprocessing Illustration



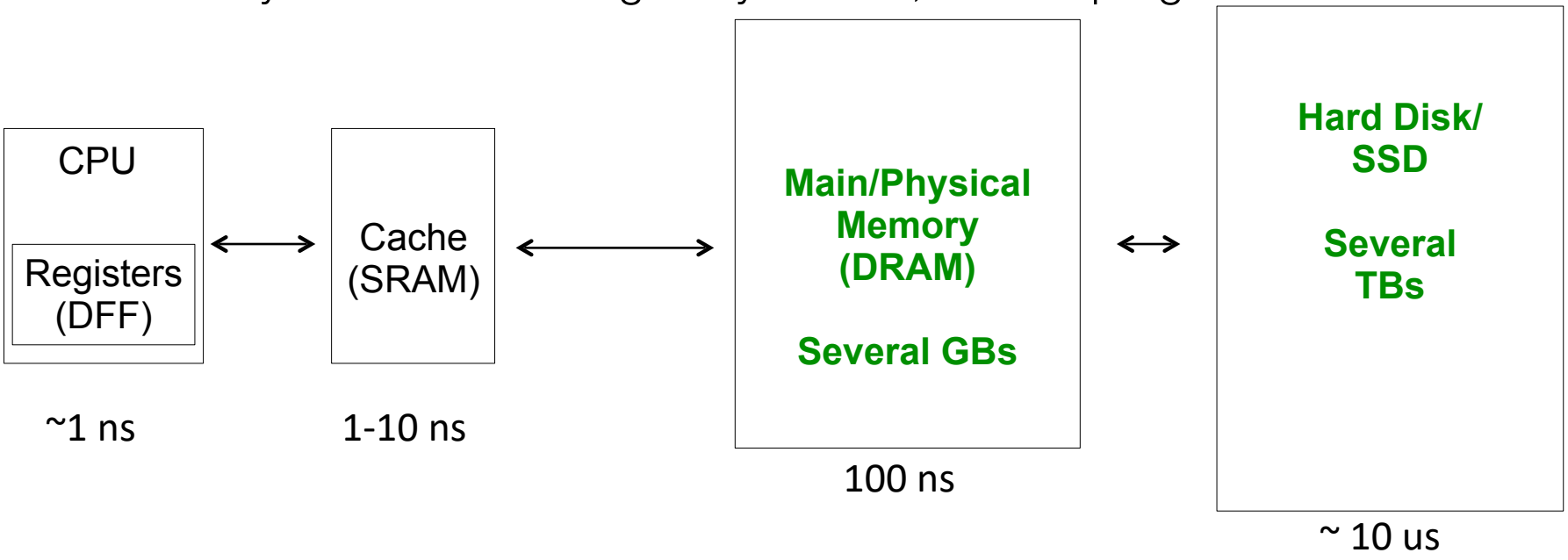
Problem 1: Space



- 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

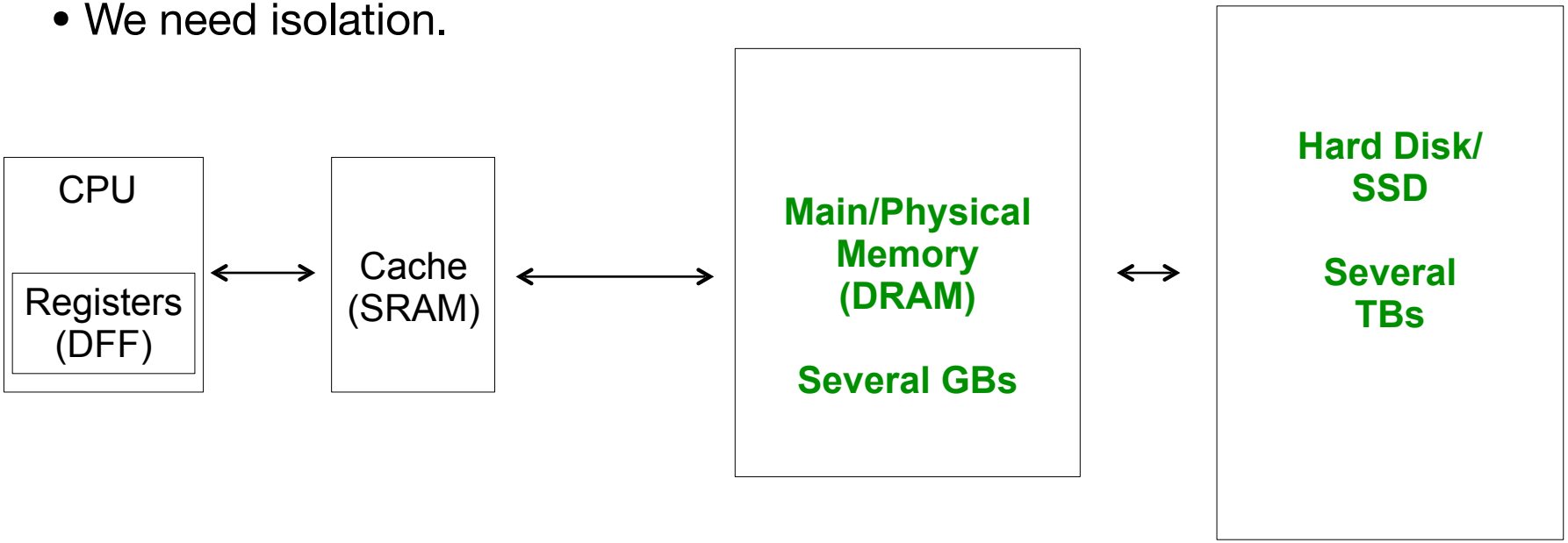
Recall: Memory Hierarchy

- 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.
- 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.



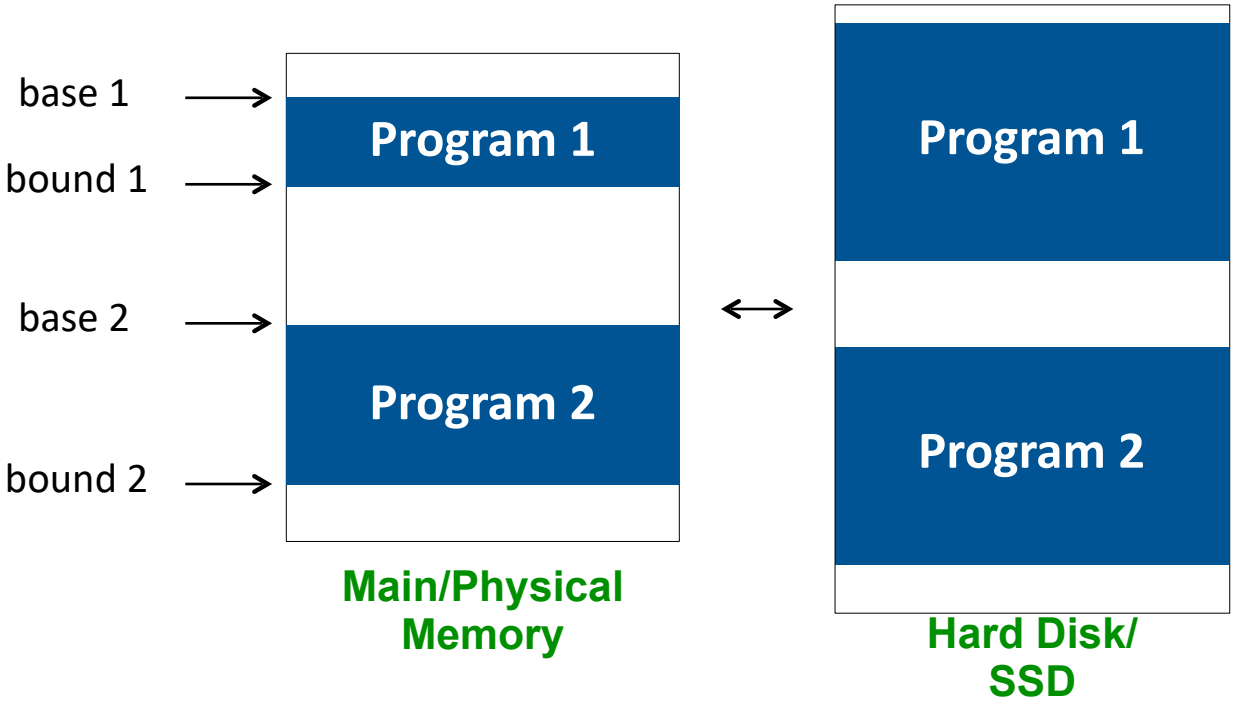
Problem 2: Security

- 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.



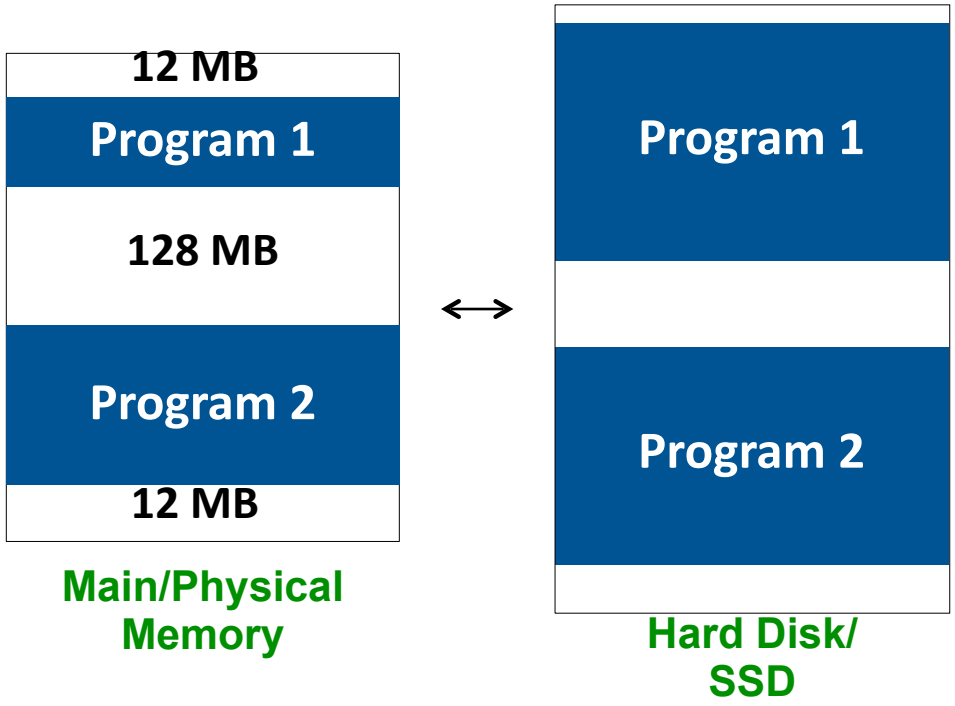
One Way to Isolate: Segments

- 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 inflexible. Makes benign sharing hard.



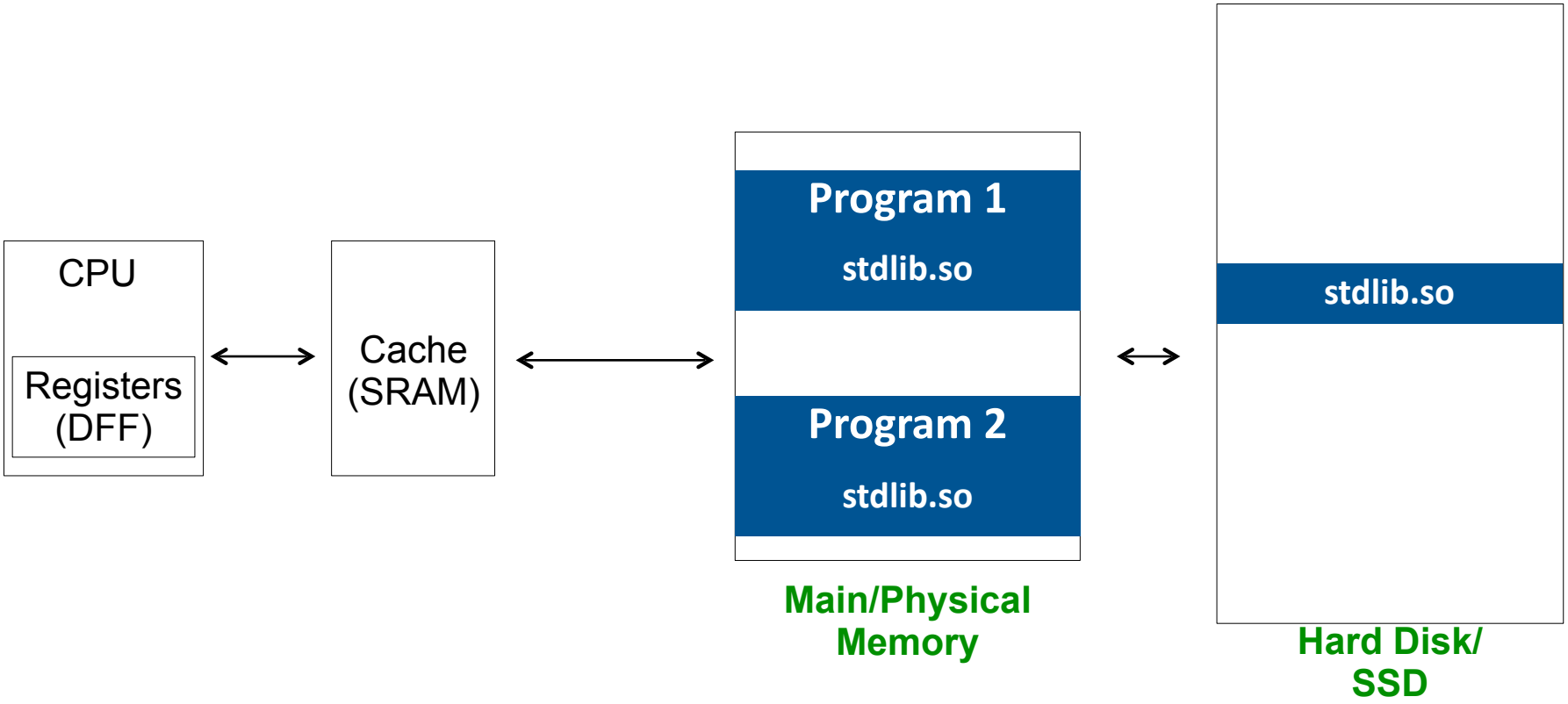
Problem 3: Fragmentation (with Segments)

- 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.

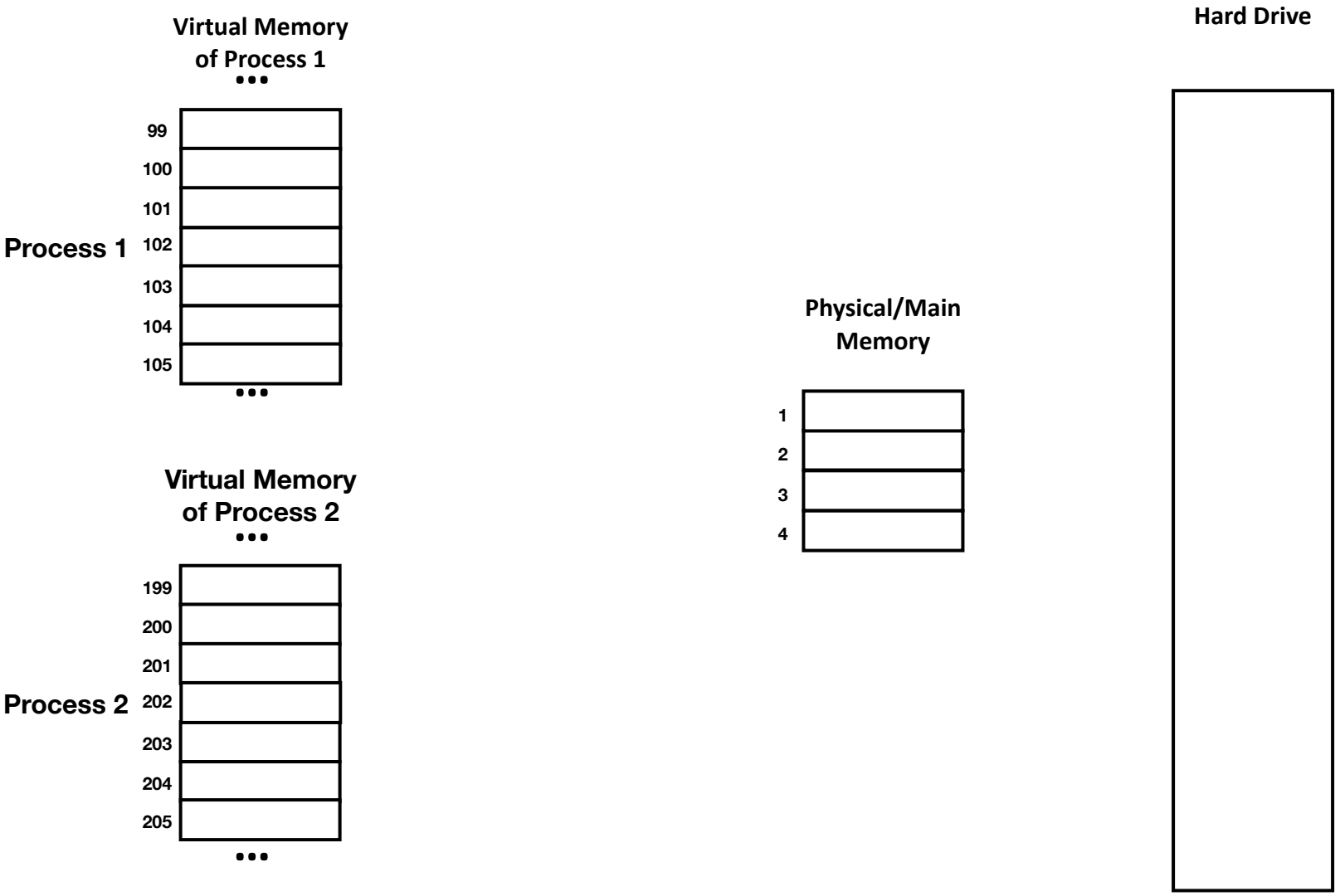


Problem 4: Benign Sharing (with Segments)

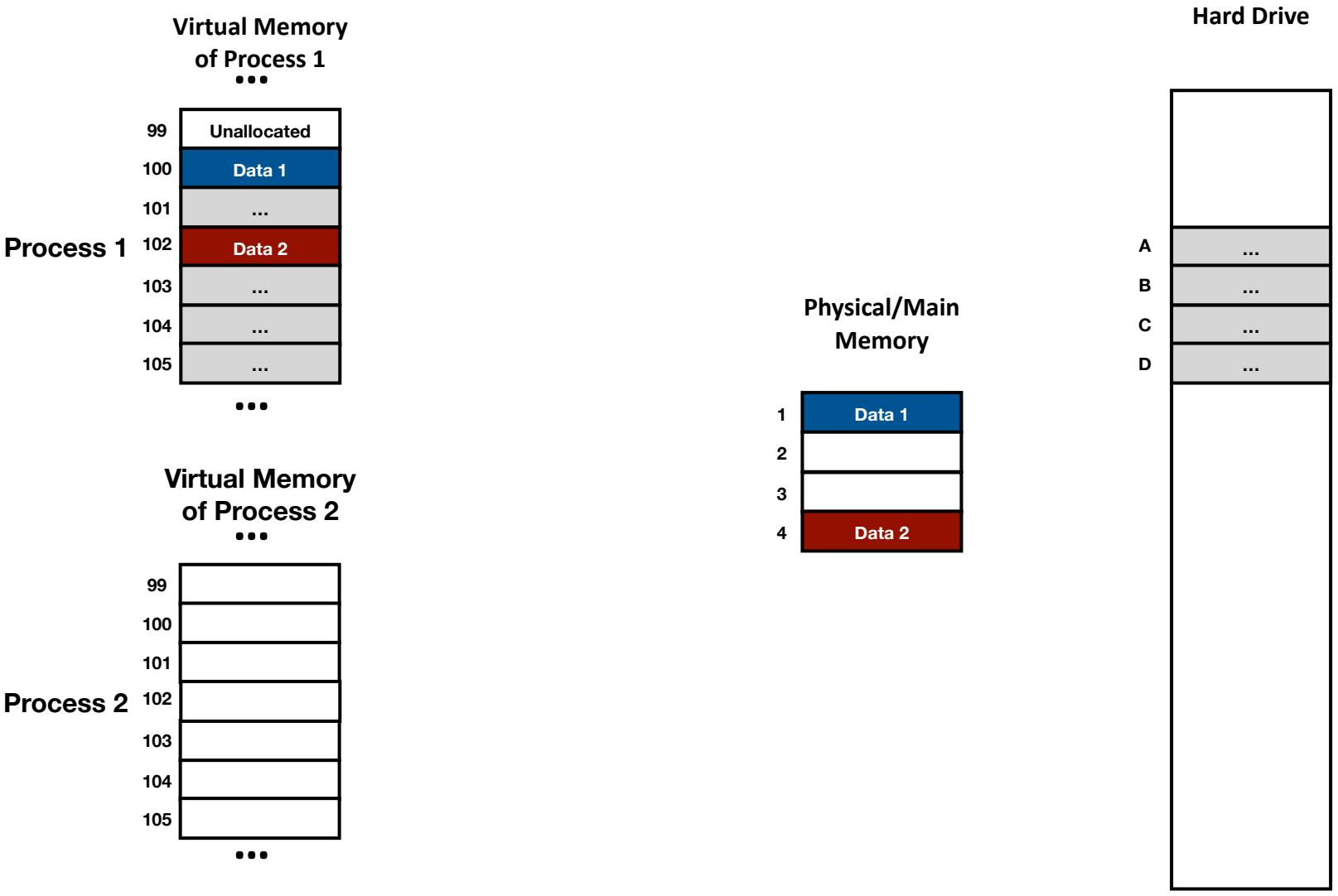
- Different programs/processes will share same data: files, libraries, etc.
- No need to have separate copies in the physical memory.
- Would be good to let other processes access part of the current's process' memory based on the "permission".



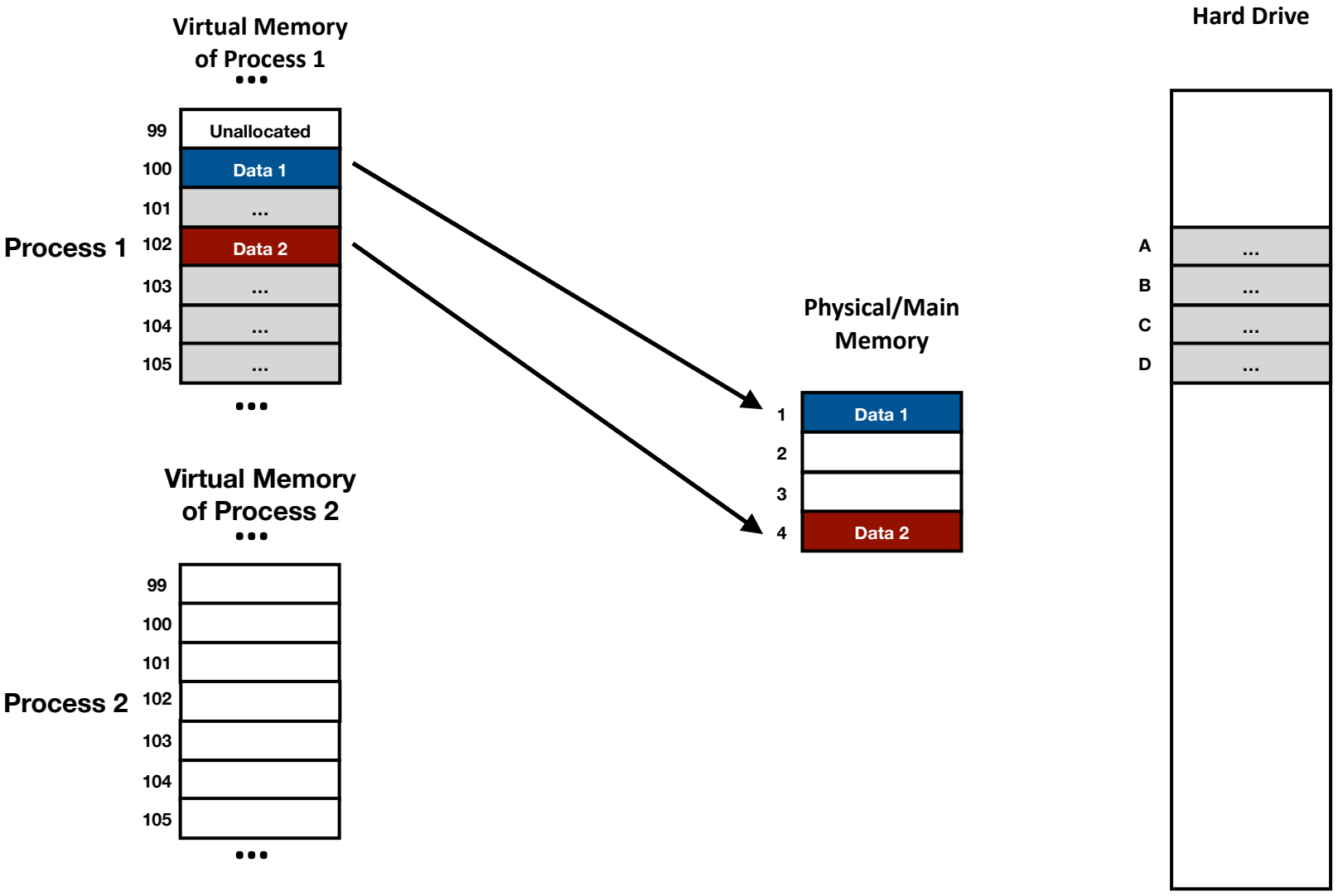
The Big Idea: Virtual Memory



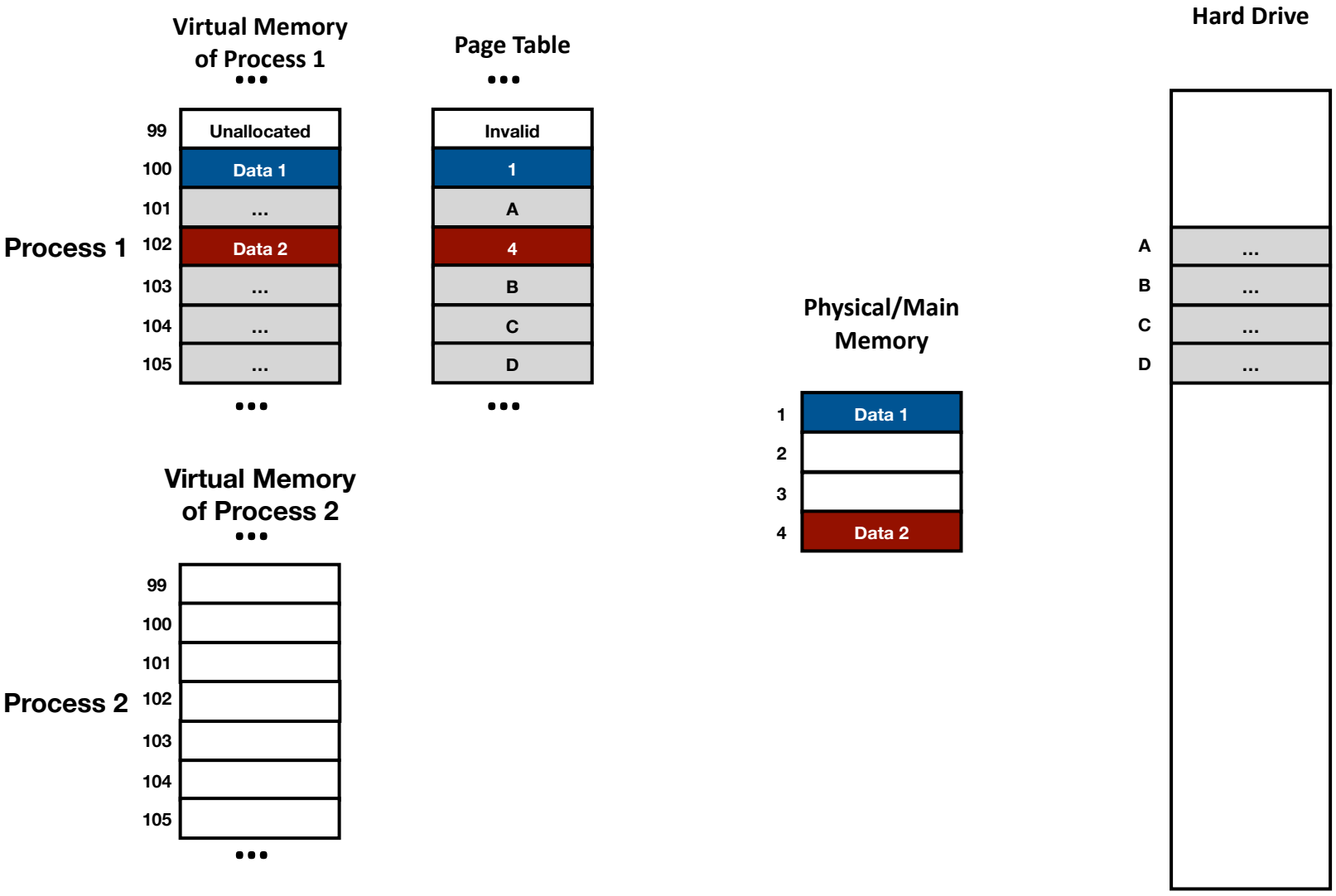
A System Snapshot



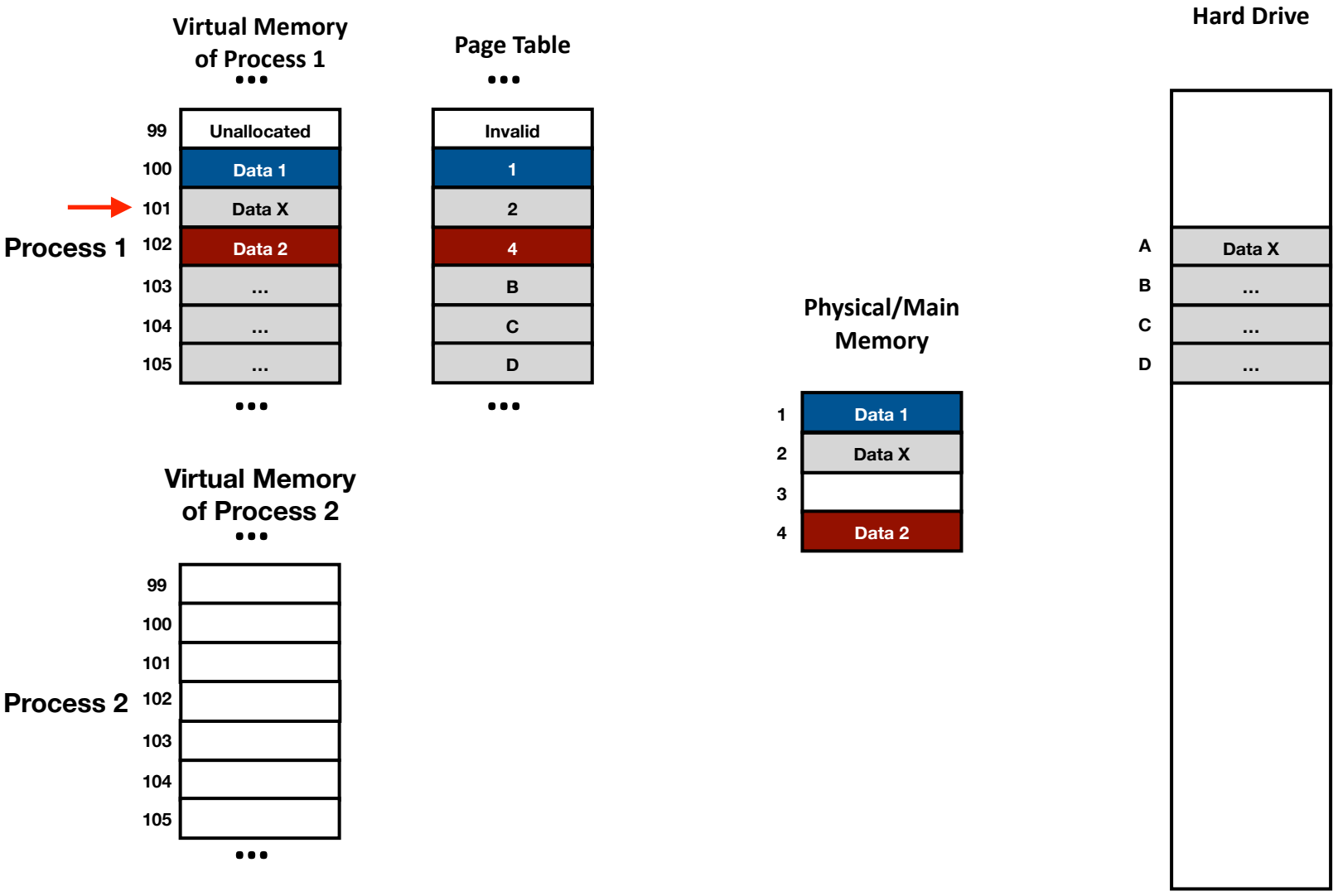
Allow Using Discontinuous Allocation



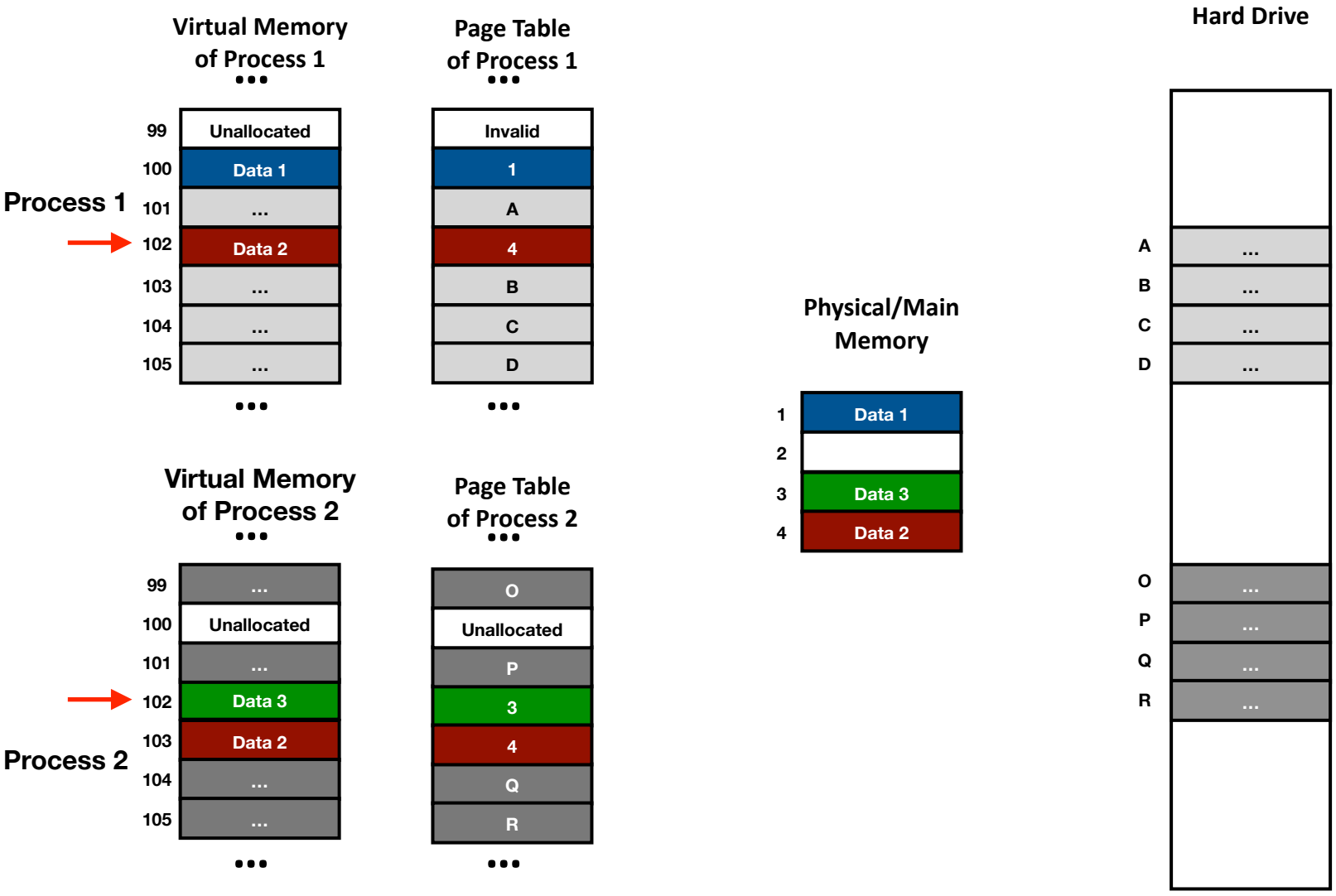
Page Table


Virtual Memory of Process 1
Page Table
Process 1
99
100
101
102
103
104
105
Unallocated
Data 1
...
Data 2
...
...
...
Invalid
1
A
4
B
C
D
Physical/Main Memory
1
2
3
4
Data 1
Data 2
Hard Drive
A
B
C
D
...
...
...
...
Virtual Memory of Process 2
Process 2
99
100
101
102
103
104
105

Demand Paging ("Caching" Data in Memory)



Prevent Unwanted Sharing



Enable Benign Sharing