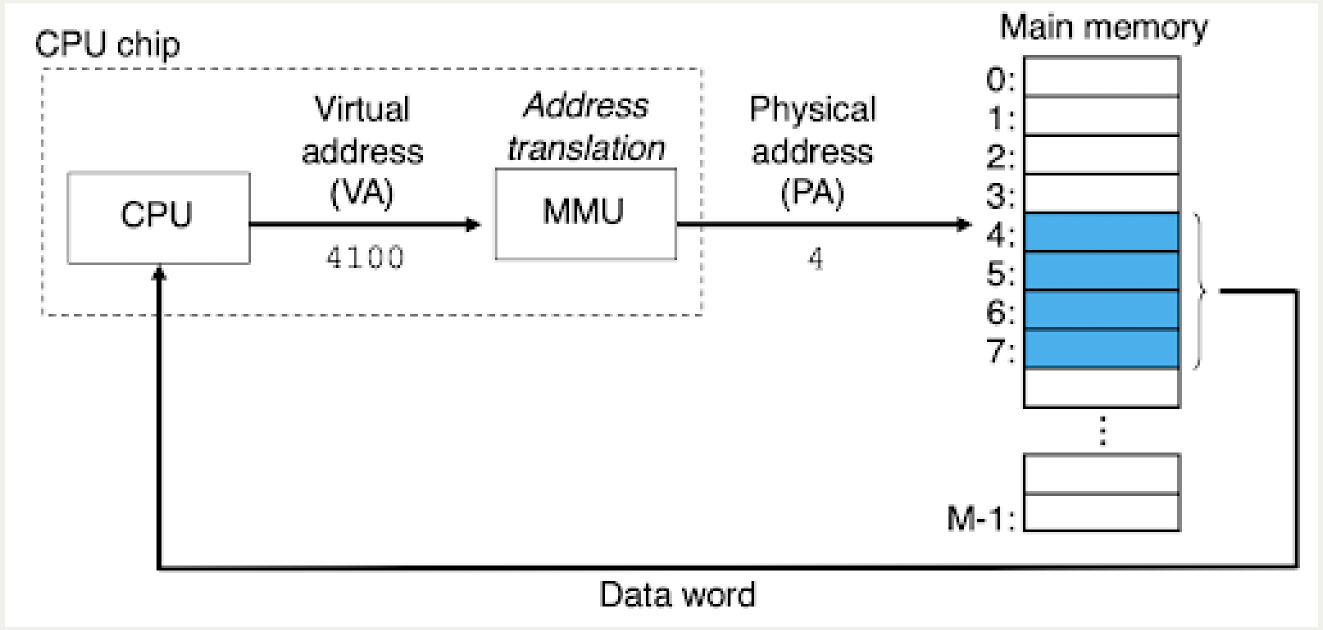
CS110 Lecture 12: Signals and Virtual Memory

CS110: Principles of Computer Systems

Winter 2021-2022

Stanford University

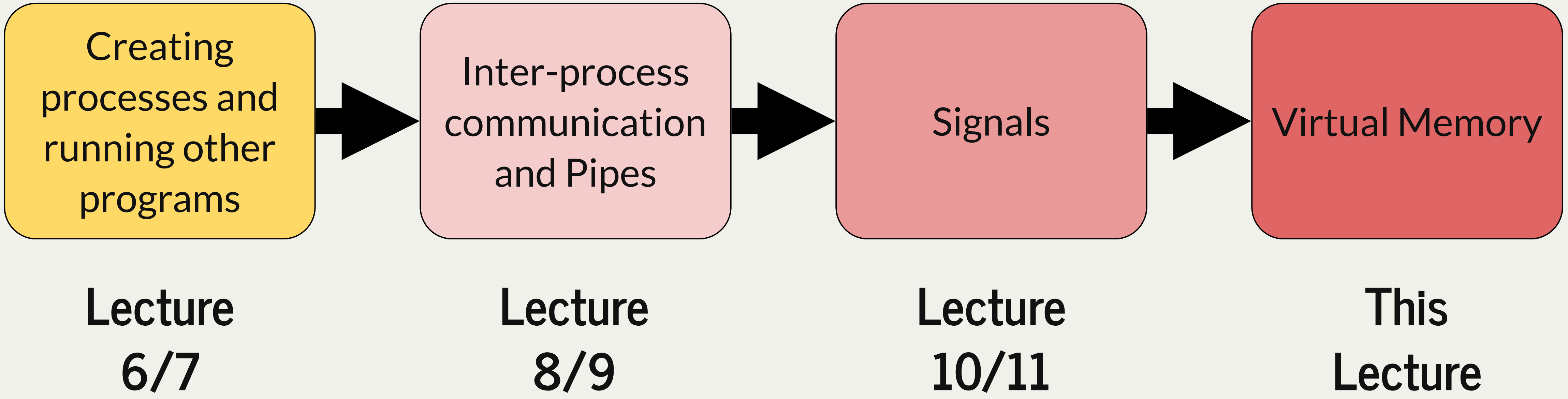
Instructors: Nick Troccoli and Jerry Cain



PDF of this presentation

<u>**CS110 Topic 2:**</u> How can our program create and interact with other programs?

Learning About Processes


Creating processes and running other programs
Inter-process communication and Pipes
Signals
Virtual Memory
Lecture 6/7
Lecture 8/9
Lecture 10/11
This Lecture

assign3: implement multiprocessing programs like "trace" (to trace another program's behavior) and "farm" (parallelize tasks)

assign4: implement your own shell!

Learning Goals

- Learn how to temporarily ignore signals with **sigprocmask** and wait for signals with **sigwait**
- Gain practice with the SIGALRM signal for timing events
- Understand how virtual memory enables multiple processes to run simultaneously

Plan For Today

- Recap: Waiting for Signals with **sigwait** and **sigprocmask**
- **Practice:** One more visit to Disneyland
- Virtual Memory

Plan For Today

- **Recap: Waiting for Signals with sigwait and sigprocmask**
- **Practice:** One more visit to Disneyland
- Virtual Memory

Waiting For Signals

- Signal handlers allow us to do other work and be notified when signals arrive. But this means the notification is unpredictable.
- A more predictable approach would be to **designate times in our program where we stop doing other work and handle any pending signals.**
  - benefits: this allows us to control when signals are handled, avoiding concurrency issues
  - drawbacks: signals may not be handled as promptly, and our process blocks while waiting
- We will not have signal handlers; instead we will have code in our main execution that handles pending signals.

Waiting For Signals

We will designate times in our program where we stop doing other work and handle any pending signals.

1. we need a way to handle pending signals
2. we need a way to turn on "do not disturb" for signals when we do not wish to handle them

Waiting For Signals

We will designate times in our program where we stop doing other work and handle any pending signals.

1. **we need a way to handle pending signals**
2. we need a way to turn on "do not disturb" for signals when we do not wish to handle them

`sigwait()`

sigwait() can be used to wait (block) on a signal to come in:

```
int sigwait(const sigset_t *set, int *sig);
```

- **set:** the location of the set of signals to wait on
- **sig:** the location where it should store the number of the signal received
- the return value is 0 on success, or > 0 on error.

Cannot wait on SIGKILL or SIGSTOP, nor synchronous signals like SIGSEGV or SIGFPE.

`sigwait()`

Here's a program that overrides the behavior for Ctl-z to print a message instead:

```c
int main(int argc, char *argv[]) {
  sigset_t monitoredSignals;
  sigemptyset(&monitoredSignals);
  sigaddset(&monitoredSignals, SIGTSTP);

  printf("Just try to Ctl-z me!\n");
  while (true) {
    int delivered;
    sigwait(&monitoredSignals, &delivered);
    printf("\nReceived signal %d: %s\n", delivered, strsignal(delivered));
  }

  return 0;
}
```

sigwait.c

sigwait()

Problem: what if the user hits Ctl-z *before we reach line 9, or between **sigwait** calls?* *It won't be handled by our code!*

```c
 1 int main(int argc, char *argv[]) {
 2   sigset_t monitoredSignals;
 3   sigemptyset(&monitoredSignals);
 4   sigaddset(&monitoredSignals, SIGTSTP);
 5
 6   printf("Just try to Ctl-z me!\n");
 7   while (true) {
 8     int delivered;
 9     sigwait(&monitoredSignals, &delivered);
10     printf("\nReceived signal %d: %s\n", delivered, strsignal(delivered));
11   }
12
13   return 0;
14 }
```

sigwait.c

sigwait()

Problem: what if the user hits Ctl-z *before we reach line 9*, *or between* ***sigwait*** *calls?* *It won't be handled by our code!*

```c
 1 int main(int argc, char *argv[]) {
 2   sigset_t monitoredSignals;
 3   sigemptyset(&monitoredSignals);
 4   sigaddset(&monitoredSignals, SIGTSTP);
 5   sleep(2);
 6 
 7   printf("Just try to Ctl-z me!\n");
 8   while (true) {
 9     int delivered;
10     sigwait(&monitoredSignals, &delivered);
11     printf("\nReceived signal %d: %s\n", delivered, strsignal(delivered));
12     sleep(2);
13   }
14 
15   return 0;
16 }
```

sigwait.c

This is a race condition: an unpredictable ordering of events where some orderings may cause undesired behavior.

Waiting For Signals

We will designate times in our program where we stop doing other work and handle any pending signals.

1. we need a way to handle pending signals
2. **we need a way to turn on "do not disturb" for signals when we do not wish to handle them**

Do Not Disturb

The **sigprocmask** function lets us temporarily block signals of the specified types. Instead, they will be queued up and delivered when the block is removed.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- **how** is **SIG_BLOCK** (add this to the list of signals to block), **SIG_UNBLOCK** (remove this from the list of signals to block) or **SIG_SETMASK** (make this the list of signals to block)
- **set** is a special type that specifies the signals to add/remove/replace with
- **oldset** is the location of where to store the *previous* blocked set that we are overwriting.

Side note: forked children inherit blocked signals! We may wish to remove a block in the child.

Do Not Disturb

Here's the same program from before, but blocking SIGTSTP as soon as possible:

```c
int main(int argc, char *argv[]) {
  sigset_t monitoredSignals;
  sigemptyset(&monitoredSignals);
  sigaddset(&monitoredSignals, SIGTSTP);
  sigprocmask(SIG_BLOCK, &monitoredSignals, NULL);

  printf("Just try to Ctl-z me!\n");
  while (true) {
    int delivered;
    sigwait(&monitoredSignals, &delivered);
    printf("\nReceived signal %d: %s\n", delivered, strsignal(delivered));
  }

  return 0;
}
```

Wait - if we call **sigwait** while signals are blocked, what happens?

Key insight: sigwait() doesn't care about blocked signals when it is called.

`sigwait.c`

Plan For Today

- Recap: Waiting for Signals with **sigwait** and **sigprocmask**
- **Practice: One more visit to Disneyland**
- Virtual Memory

Revisiting Disneyland (Again)

Let's rewrite our **five-children.c** program using **sigwait** instead of signal handlers.

1. Turn on "do not disturb" to block signals until we are ready
2. Spawn the child processes (**important to do this *after* blocking SIGCHLD!**)
3. Call **sigwait** in a loop to wait for incoming signals that children have finished

Signal Block are Inherited by Children

Key Idea: signal blocks are inherited by child processes. Therefore we must unblock in the child - otherwise, e.g. if we are execvp'ing, that program won't receive SIGCHLD!

```c
static void spawnChildren(size_t numChildren, sigset_t signalsToUnblock) {
  for (size_t kid = 1; kid <= numChildren; kid++) {
    if (fork() == 0) {
      sigprocmask(SIG_UNBLOCK, &signalsToUnblock, NULL);
      sleep(3 * kid); // sleep emulates "play" time
      printf("Child #%zu tired... returns to parent.\n", kid);
      exit(0);
    }
  }
}
```

Revisiting Disneyland (Again)

Let's rewrite our **five-children.c** program using **sigwait** instead of signal handlers.

1. Turn on "do not disturb" to block signals until we are ready
2. Spawn the child processes (**important to do this *after* blocking SIGCHLD!**)
3. Call **sigwait** in a loop to wait for incoming signals that children have finished

Problem: if we have code like this, the parent will not wake up unless a child returns:

```c
while (numChildrenParentSees < kNumChildren) {
    int delivered;
    sigwait(&monitoredSignals, &delivered);

    // doesn't work - parent may sleep for longer including sigwait,
    // and parent won't hear signals while snoozing.
    snooze(5);
  }
```

SIGALRM

Let's rewrite our **five-children.c** program using **sigwait** instead of signal handlers.

Idea: we can't sleep ourselves anymore. Let's use *another signal* to keep track of our "sleep time" instead; like an alarm clock.

Solution: there is a signal SIGALRM we can send to ourselves X seconds later.

```c
// This function schedules a SIGALRM signal to be sent to us in 'duration' seconds.
static void setAlarm(double duration) {
  int seconds = (int)duration;
  int microseconds = 1000000 * (duration - seconds);
  struct itimerval next = {{0, 0}, {seconds, microseconds}};
  setitimer(ITIMER_REAL, &next, NULL);
}
```

Revisiting Disneyland (Again)

Let's rewrite our **five-children.c** program using **sigwait** instead of signal handlers.

1. Turn on "do not disturb" to block SIGCHLD and SIGALRM signals until we are ready
2. Spawn the child processes (**important to do this *after* blocking SIGCHLD!**)
3. Set an alarm for 5 seconds from now
4. Loop until we have seen all children return:
    1. listen for incoming SIGCHLD or SIGALRM signals
    2. if **SIGCHLD**: clean up the child
    3. if **SIGALRM**: "wake up", count cleaned up children, set another alarm if more still playing

Demo: five-children-sigwait.c

`five-children-sigwait-soln.c`

Overview: Signals and Concurrency

- Concurrency is powerful: it lets our code do many things at the same time
- It can run faster (more cores!), and can do more (run many programs in background)
- Signals are a way for concurrent processes to interact
- Send signals with **kill** and **raise**
- Making sure code running in a signal handler works correctly is difficult
- An alternative approach for signals is to block signals until designated times when we wait for them
- *Race conditions* occur when code can see data in an intermediate and invalid state
- Assignments 3 and 4 use signals, as a way to start easing into concurrency before we tackle multithreading
- Take CS149 if you want to learn how to write high concurrency code that runs 100x faster

Plan For Today

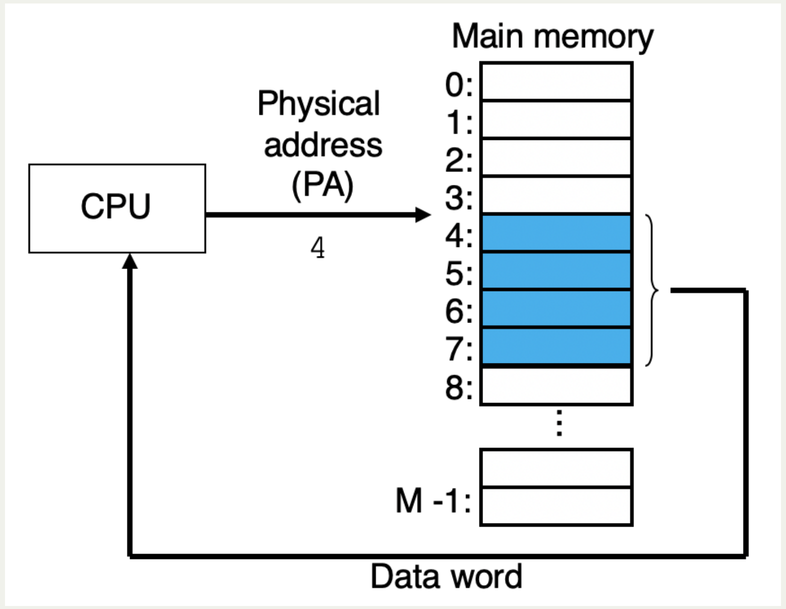
- Recap: Waiting for Signals with **sigwait** and **sigprocmask**
- **Practice:** One more visit to Disneyland
- **Virtual Memory**

Virtual Memory

- Memory (RAM, "Random access memory") is where relevant program data (stack, heap, etc.) is stored. It is a contiguous array of bytes.
- When multiple processes run concurrently, they all may need a portion of memory
- The Operating System manages memory use and access
- **Core question:** how does the OS manage memory so multiple processes can use it?

Before Virtual Memory

- Before multiprocessing, one process could run at a time
- Used "physical addressing"; addresses used by programs and the OS were real addresses in physical memory.
- A **physical address** is an address in physical memory.
- A process owns all of memory while it's running - can access any address



The Need for Virtual Memory

Challenges with multiple processes running:

- how do we partition memory?
- what if one process accesses the memory of another?
- what if we run out of physical memory?

Virtual Memory Core Idea: have the processes work with virtual (”fake”) addresses. The OS will decide what physical addresses they actually are.

Virtual Memory

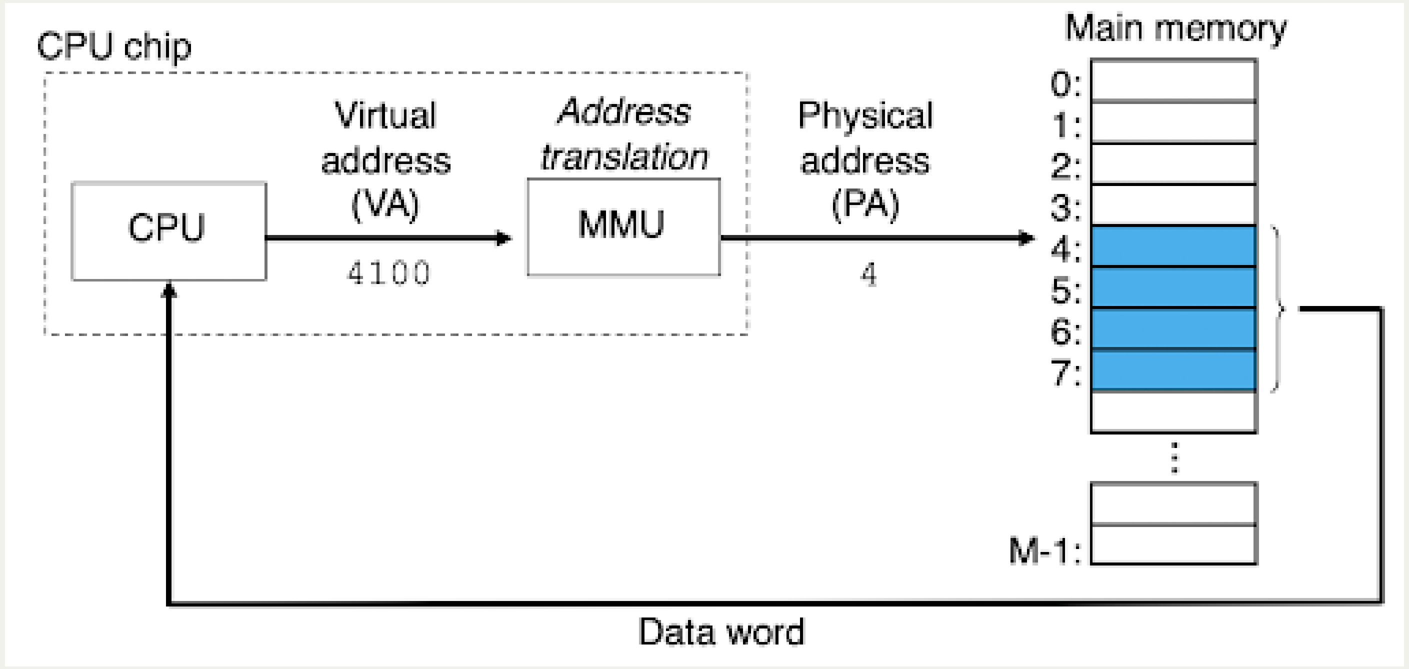
Challenges with multiple processes running:

- how do we partition memory? **-> The OS decides as it goes**
- what if one process accesses the memory of another? **-> Virtual address spaces are separate and monitored by OS**
- what if we run out of physical memory? **-> OS can play tricks to swap memory to disk when needed, and map addresses only on demand.**

The Memory Management Unit

We need an extremely fast way to convert virtual addresses to physical addresses.

- The Memory Management Unit ("MMU") is a special chip in the CPU that does this.
- *extremely fast* - otherwise impacts performance!



Virtual vs. Physical Address Spaces

- An **address space** is an ordered sequence of integer addresses, starting at 0.
- The **physical address space size** is limited by hardware (how much RAM you have)
- The **virtual address space size** is limited by pointer size only (64-bit on myth)

Virtual Memory

Challenges with multiple processes running:

- **<u>how do we partition memory? -> The OS decides as it goes</u>**
- what if one process accesses the memory of another? **-> Virtual address spaces are separate and monitored by OS**
- what if we run out of physical memory? **-> OS can play tricks to swap memory to disk when needed, and map addresses only on demand.**

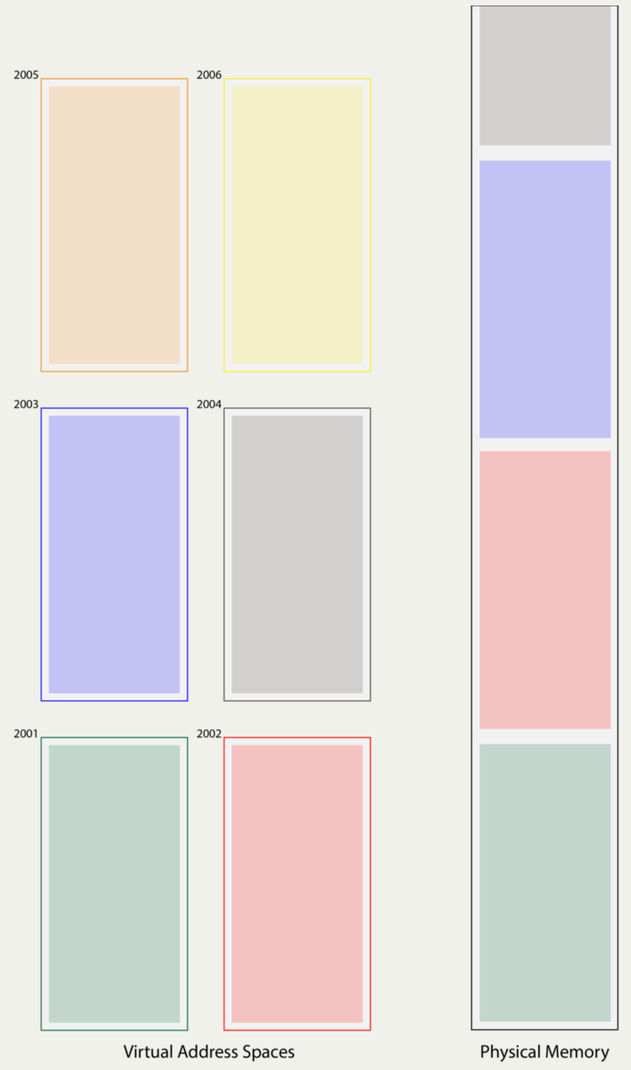
Mapping Virtual to Physical, Attempt #1

How do we map virtual addresses to physical addresses for each process?

One idea: map each process's virtual address space to physical memory starting at some offset. Fast translation!

Problems:

- This assumes a **ton** of physical memory
- Perhaps much of a process's virtual address space is unused. Here, we reserve physical space anyway.



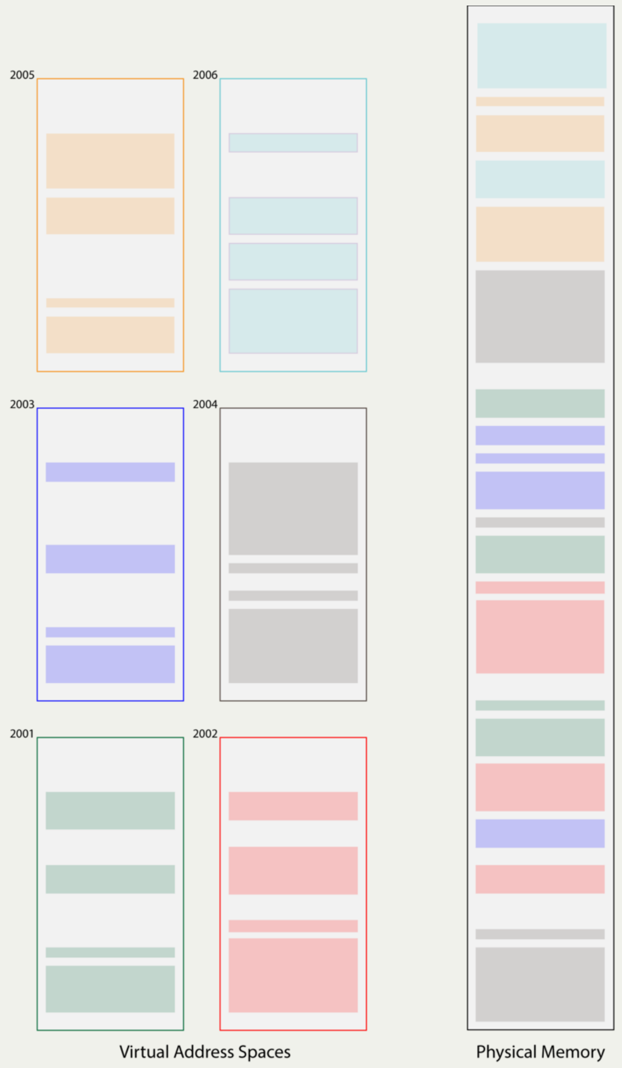
Mapping Virtual to Physical, Attempt #2

How do we map virtual addresses to physical addresses for each process?

Another idea: just map each process's segments to physical memory. That way we only map what is allocated.

Problems:

- More complicated address translation
- Memory segments may grow (stack, heap)
- Physical memory could become fragmented with variable-sized mappings



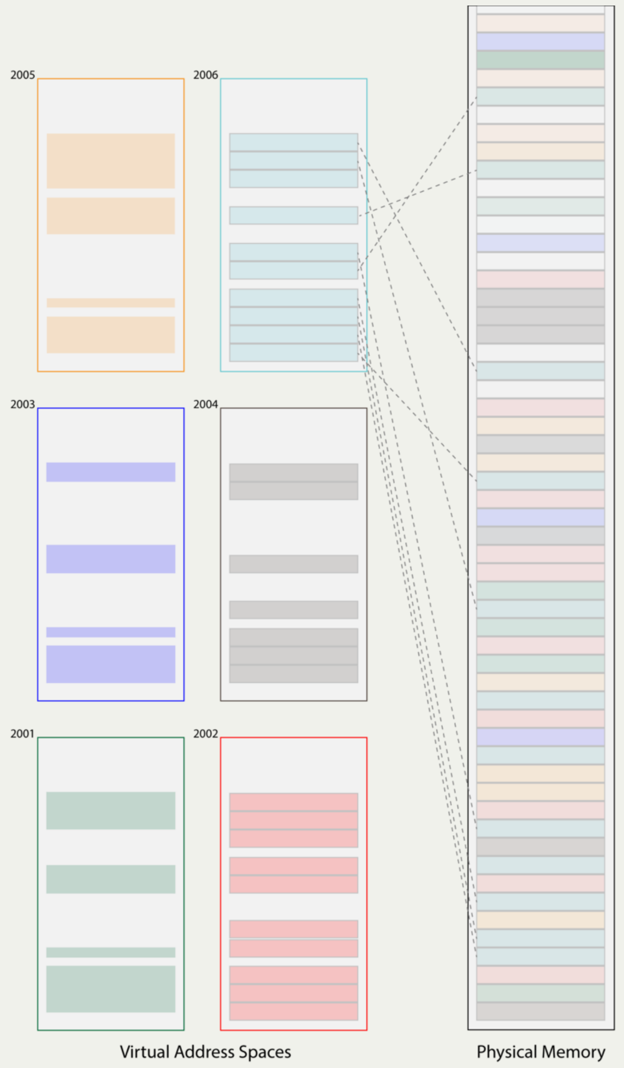
We need to define a standard ”unit” of memory that is mapped at a time, so that mappings aren’t variable size.

Mapping Virtual to Physical, Attempt #3

How do we map virtual addresses to physical addresses for each process?

A third idea: map memory only as needed, in units of pages (a page is 4096 bytes or some other power of 2).

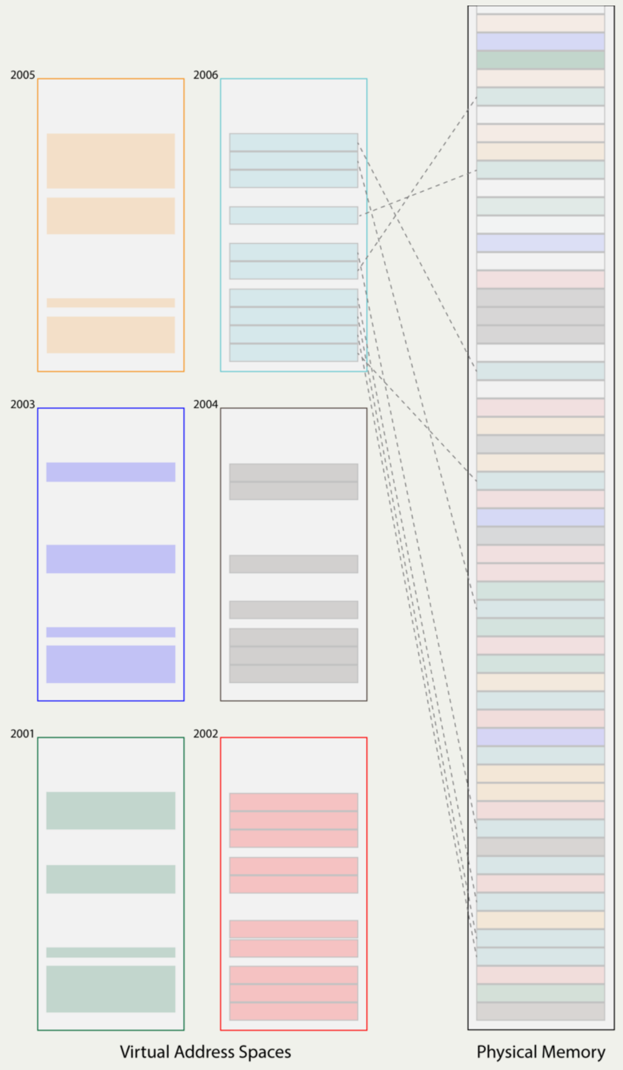
- memory segments are chopped up into pages and pages are allocated as needed.
- helps with fragmentation; all pages are the same size.
- more pages can be allocated on demand as needed.
- the OS has a **page table** mapping virtual pages -> physical ones



Mapping Virtual to Physical, Attempt #3

How do we map a virtual page to a physical one?

- Virtual address is concatenation of a *virtual page number* (which virtual page it is in) and *virtual page offset* (where in that page it is)
- The MMU can go from virtual page number to physical page number.
- The corresponding physical address is the **physical page number** plus the *virtual page offset* (since virtual and physical pages are the same size!)



Mapping Virtual to Physical, Attempt #3

How do we map a virtual page to a physical one?

- Here's an example of a page table for a single process:

7FFFFFFF80234 maps to 835432

0000DDEE65111 maps to 45D834

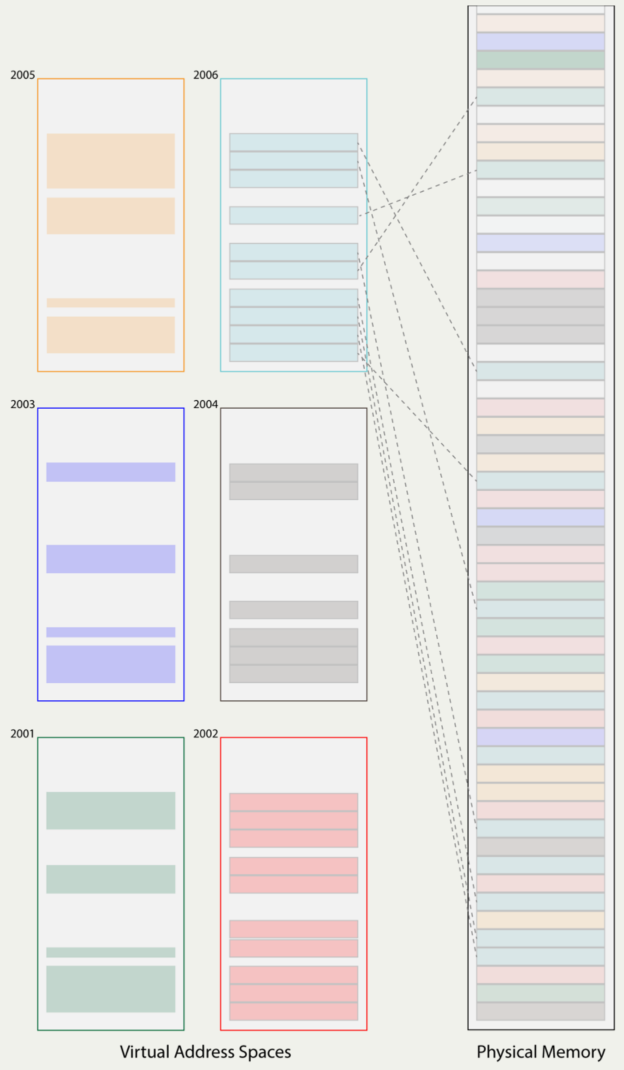
many additional entries

0000A9CF9AAAF maps to 12387B

- Translation example for virtual address **0x7FFFFFFF80234230**:

0x7FFFFFFF80234230 & 0xFFF = 0x230

0x230 | (0x835432 << 12) = 0x83543223



Virtual Memory

Challenges with multiple processes running:

- how do we partition memory? **-> The OS decides as it goes**
- **<u>what if one process accesses the memory of another? -> Virtual address spaces are separate and monitored by OS</u>**
- what if we run out of physical memory? **-> OS can play tricks to swap memory to disk when needed, and map addresses only on demand.**

Virtual Address Space Isolation

What if one process accesses the memory of another? **-> Virtual address spaces are separate and monitored by OS**

- Each process has its own virtual address space; this is what all processes see when they are running (and what we see in GDB!)
- when a process accesses an address, it goes through the OS
- if a memory address is invalid, OS tells the process
- mappings are kept for each process; isolated address spaces

Virtual Memory

Challenges with multiple processes running:

- how do we partition memory? **-> The OS decides as it goes**
- what if one process accesses the memory of another? **-> Virtual address spaces are separate and monitored by OS**
- what if we run out of physical memory? **-> OS can play tricks to swap memory to disk when needed, and map addresses only on demand.**

Virtual Memory

What if we run out of physical memory? **-> OS can play tricks to swap memory to disk when needed, and map addresses only on demand.**

- Virtual address spaces make it appear to processes like there is more memory than there actually is
- OS maps pages on demand
- OS can *evict pages to the hard disk* when it needs more space in physical memory
- Other ideas to make more memory space available: e.g. Mac memory compression
- Weird idea: main memory is a cache for the hard disk

Virtual Memory

- Virtual memory is an extremely powerful **abstraction** of memory for processes
- Processes have no idea that this system is in place - completely invisible
- Gives the operating system flexibility in how to best manage memory
- MMU enables fast mappings

Recap

- Recap: Waiting for Signals with **sigwait** and **sigprocmask**
- **Practice:** One more visit to Disneyland
- Virtual Memory

Next time: Introduction to multithreading