

Based on slides by Harsha V. Madhyastha

EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 14: Process creation and fork

Nicole Hamilton

[https://web.eecs.umich.edu/~nham/
nham@umich.edu](https://web.eecs.umich.edu/~nham/nham@umich.edu)

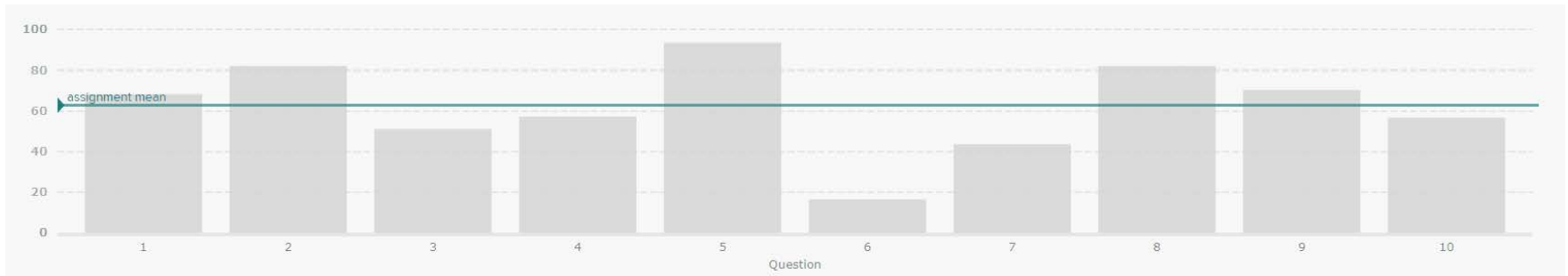
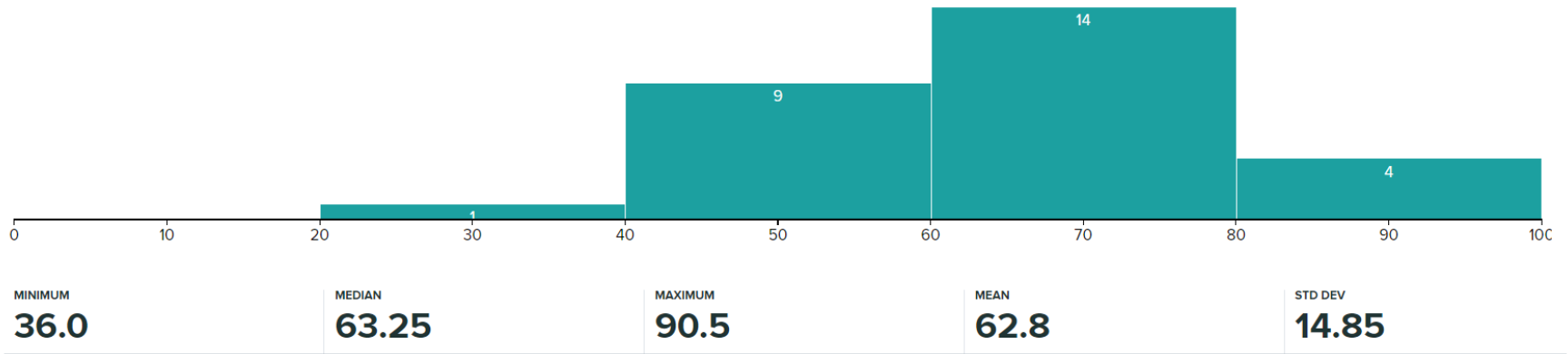
Agenda

1. Midterm.
2. Review of MLP and kernel vs. user.
3. Process creation and fork.
4. Project 3 due July 27.







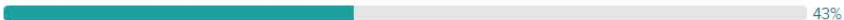



Agenda

1. **Midterm.**
2. Review of MLP and kernel vs. user.
3. Process creation and fork.
4. Project 3 due July 27.

Midterm



Regrade requests must be submitted by 11:59 pm EDT tonight.

QUESTION	POINTS	MEAN
1: Mesa vs. Hoare semantics, stolen and spurious wakeups Click to Add Tags	5 points	 68%
2: Disable interrupts, then spin lock Click to Add Tags	5 points	 82%
3: Timer interrupt handler Click to Add Tags	5 points	 51%
4: What happens when a thread returns? Click to Add Tags	5 points	 57%
5: RAIL in the context of locks Click to Add Tags	5 points	 93%
6: Optimizing the lock Click to Add Tags	5 points	 16%
7: Global order Click to Add Tags	5 points	 43%
8: Deadlock sequence Click to Add Tags	5 points	 82%
9: Wait for me! Click to Add Tags	30 points	 70%
10: Safety First Click to Add Tags	30 points	 56%

Optimizing the lock

You were asked to optimize this lock given that contention was expected to be very low.

The insight it called for was to notice if you had a TestAndSet that was safe to use on the guard, it would also be safe to use on the status variable.

Solution is to attempt a TestAndSet of the status variable and only do the full lock routine if it failed.

```
lock( )
{
    disable interrupts;
    while ( TestAndSet( guard ) )
        ;
    if ( status == FREE )
        status = BUSY;
    else
    {
        add thread to queue of threads
        waiting for lock;
        switch to next ready thread;
    }
    guard = 0;
    enable interrupts;
}
```

Optimizing the lock

This is a partial solution.

What's wrong with this?

There's still a race to check and then set the status.

```
lock( )
{
    if ( !TestAndSet( status ) )
        return;
    disable interrupts;
    while ( TestAndSet( guard ) )
        ;
    if ( status == FREE )
        status = BUSY;
    else
    {
        add thread to queue of threads
        waiting for lock;
        switch to next ready thread;
    }
    guard = 0;
    enable interrupts;
}
```

Optimizing the lock

This is all it takes.

A possibly better solution might be to loop, attempting a small number of TestAndSet operations before doing the full lock routine.

```
lock( )
{
    if ( !TestAndSet( status ) )
        return;
    disable interrupts;
    while ( TestAndSet( guard ) )
        ;
    if ( TestAndSet( status ) )
    {
        add thread to queue of threads
        waiting for lock;
        switch to next ready thread;
    }
    guard = 0;
    enable interrupts;
}
```


Global order

Consider a process with 5 threads, 1 through 5, that must share 5 resources, A through E, making requests as follows.

1. BAC
2. BAD
3. EAC
4. DE
5. AD

Step	Order
1. BAC	BAC
2. BAD	BA[CD]
3. EAC	[BE]A[CD]
4. DE	<i>inconsistent!</i>
5. AD	[BE]A[CD]

Solution is for thread 4 to unlock D, then take E, then D in the correct global order.

Agenda

1. Midterm.
2. Review of MLP and kernel vs. user.
3. Process creation and fork.
4. Project 3 due July 27.

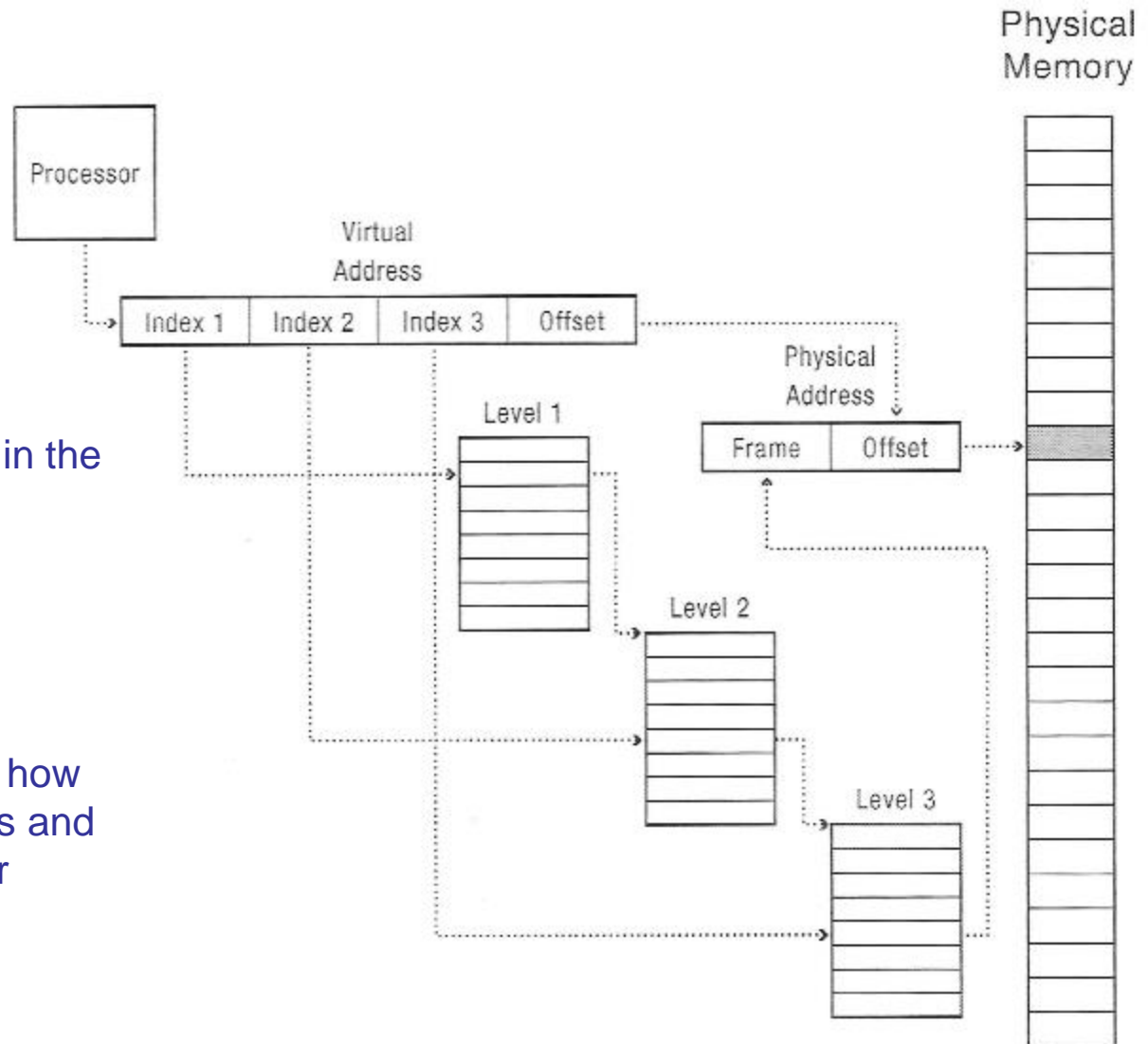
Multi-level paging

Correction

Multi-level paging is managed by software in the OS.

But accesses for TLB misses are hardware-accelerated.

The hardware defines how many levels it supports and which bits are used for indexing each level.



System calls

When you call `cin` in your C++ program:

1. `cin` calls `read()`, which executes assembly-language instruction `syscall`.
2. `syscall` traps to kernel at pre-specified location.
3. kernel's `syscall` handler calls kernel's `read()`.

To handle trap to kernel, hardware atomically:

1. Sets the mode bit to kernel.
2. Saves registers, PC, SP.
3. Changes SP to kernel stack.
4. Changes to the kernel's address space.
5. Jumps to exception handler.

Arguments to system calls

Two options:

1. Store in registers.
2. Store in memory (**in whose address space?**)

Kernel first checks validity of arguments:

```
read( int fd, void *buf, size_t size )
```

Is fd a valid descriptor for open file?

Are all addresses in [buf, buf+size) valid?

Are all addresses in [buf, buf+size) writable?

Protection summary

Safe to switch from user to kernel mode because control is only transferred to certain locations and the number of entry points to the system is kept limited to limit the *attack surface*.

Where are these locations stored?

Interrupt vector table.

Who can modify interrupt vector table?

Only the kernel. Set once at boot, then never changed.

Why is it easier to control access to interrupt vector table than mode bit?

The mode bit changes constantly and every change must be scrutinized. But the IVT never changes.

Address Space Protection

How are address spaces protected?

Separation of translation data, meaning the translation data must be protected.

How is translation data protected?

Can update translation data only if mode bit set.

How is mode bit protected?

Sets/reset mode bit when transitioning from user-level to kernel-level code and back.

Transitions limited by interrupt vector table.

Protection boils down to init process which sets up interrupt vector table when system boots up.

Memory Management so far

How to represent virtual address spaces?

Larger than physical memory.

Controlled sharing.

Independently grow parts of address space.

Low overhead for sparse address spaces.

Paging.

How to minimize hardware interface?

Manipulate protection bits to incur page faults and maintain additional state in kernel.

How to separate kernel vs. user address space?

Pinning of pages, map physical memory into pager.

Agenda

1. Midterm.
2. Review of MLP and kernel vs. user.
3. **Process creation and fork.**
4. Project 3 due July 27.

Process creation

Steps

1. Allocate process control block.
2. Initialize translation data for new address space.
3. Read program image from executable into memory.
4. Initialize registers.
5. Set mode bit to “user”.
6. Jump to start of program.

Need hardware support for the last few steps.

We're undoing the switch from user to kernel that happens in a system call.

Processes sharing memory

How to divide physical memory among processes?

Should every process get same amount of memory?

Fairness versus efficiency.

Global replacement

Can evict pages from the faulting process or any other.

Local replacement

Can evict pages only from the faulting process.

Must determine how many frames each process gets.

Pros and cons?

Thrashing

What happens if many large processes all actively use their entire address space?

Performance degrades rapidly as miss rate goes up.

Average access time = hit rate * hit time + miss rate * miss time

Example: Assume hit time = 100 ns = .0001 ms, miss time = 10 ms.

Average access time (100% hit rate) = .0001 ms

Average access time (1% miss rate) = .100099 ms

Average access time (10% miss rate) = 1.00090 ms

Solutions to Thrashing

Buy more DRAM.

Very common solution in cloud servers.

Price per GB fallen by 4x since 2009.

Run fewer processes for longer time slices.

Reduces page faults.

But may cause poor interactivity due to long time slices.

Working set

Thrashing depends on portion of address space actively used by each process.

What do we mean by “actively using”?

Working set = all pages used in last T seconds.

Larger working set → needs more memory to run well.

Sum of all working sets should fit in memory.

One solution is to run a subset of the processes that fit in memory.

How to measure size of working set?

Periodic sweep of clock hand in LRU clock.

Unix process creation

System uses a sequence of two calls to start a process:

1. `fork()` creates a copy of current process.
2. `exec(program, args)` replaces current address space with specified program.

Why first copy the process only to overwrite it?

Allows sharing of code, file descriptors, other state information and results in a simple interface.

Windows by contrast, uses a single `CreateProcess()` system call, but requires a very complex set of arguments to deal with all the possible cases.

Windows CreateProcess

There is no fork().

Creates the child running a new executable, returns a handle to the child.

argv is passed as a string, not an array.

Child process retrieves the command line with GetCommandLine(). C runtime turns that into argc, argv.

Slightly complex rules for words containing spaces or quotes.

Lots of options for debugging, etc.

Two versions.

```
BOOL CreateProcessA(  
    LPCSTR          lpApplicationName,  
    LPSTR           lpCommandLine,  
    LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    BOOL           bInheritHandles,  
    DWORD          dwCreationFlags,  
    LPVOID         lpEnvironment,  
    LPCSTR         lpCurrentDirectory,  
    LPSTARTUPINFOA lpStartupInfo,  
    LPPROCESS_INFORMATION lpProcessInformation  
);
```

```
BOOL CreateProcessW(  
    LPCWSTR         lpApplicationName,  
    LPWSTR          lpCommandLine,  
    LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    BOOL           bInheritHandles,  
    DWORD          dwCreationFlags,  
    LPVOID         lpEnvironment,  
    LPCWSTR        lpCurrentDirectory,  
    LPSTARTUPINFOW lpStartupInfo,  
    LPPROCESS_INFORMATION lpProcessInformation  
);
```


Unix process creation

System uses a sequence of two calls to start a process:

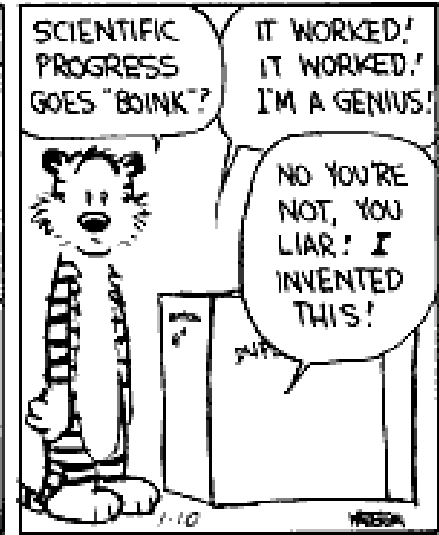
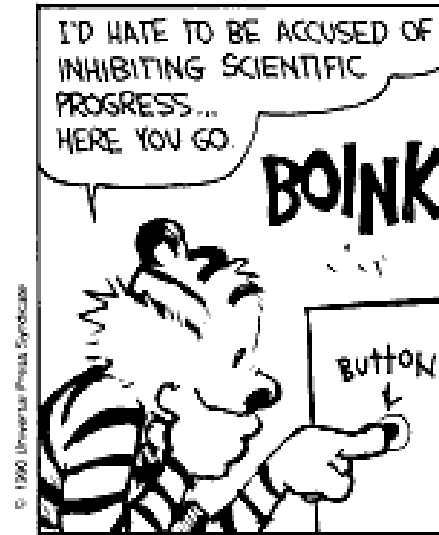
1. `fork()` creates a copy of current process.
2. `exec(program, args)` replaces current address space with specified program.

Any problems with child being an *exact* clone of parent?

Cloning



BROTHER! YOU DOUBTING THOMASES GET IN THE WAY OF MORE SCIENTIFIC ADVANCES WITH YOUR STUPID ETHICAL QUESTIONS! THIS IS A **BRILLIANT** IDEA! HIT THE BUTTON, WILL YA?



Need to know which is the parent and which is the child.

Fork and exec

Fork uses the return code to differentiate parent from child.

Child gets return code 0.

Parent gets child's unique process id or *pid*.

```
int pid = fork( );
if ( pid == 0 )
{
    exec (); /* child */
}
else
{
    /* parent */
}
```

Fork bombs

Can easily overwhelm the system by creating thousands of forked copies with malicious code.

In C++:

```
for ( int i = 100; i--; )  
    fork( );
```

In bash:

```
:( ){ :|:& } ; :
```

Subtleties in handling fork

Buggy code from autograder that caused false complaints.

What is the race condition here?

Initially, the child's address space is a copy of the AG, which is huge.

```
if ( !fork( ) )  
    exec( command );
```

```
While ( child is alive )  
    if ( size of child address space > max )  
    {  
        print "process took too much memory";  
        kill child;  
        break;  
    }
```

Avoiding work on fork

Copying entire address space is expensive

Instead, Unix uses `copy-on-write`.

Maintain reference count for each physical page.

On `fork()`, copy only the page table of parent.

Increment reference count by one.

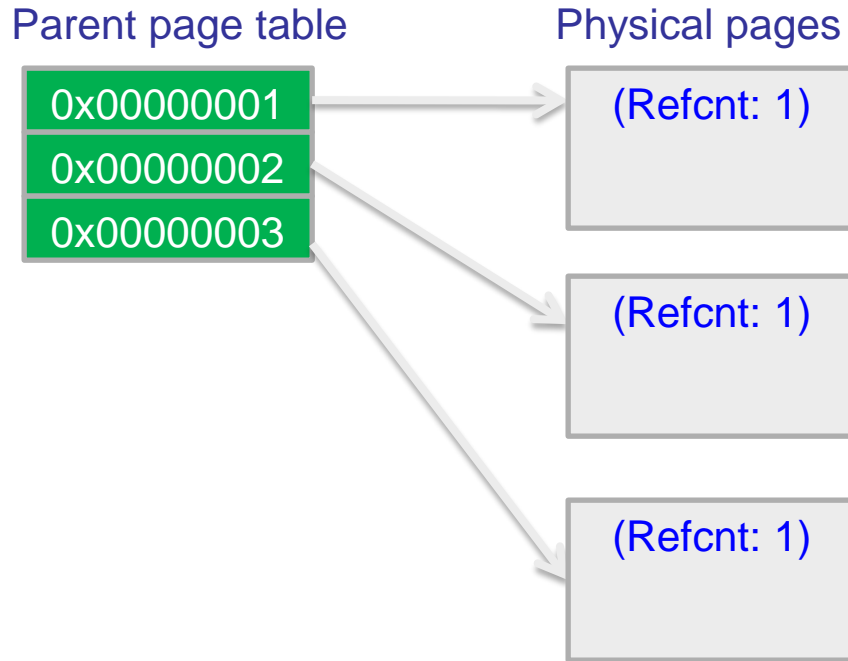
On store by parent or child to page with `refcnt > 1`:

Make a copy of the page with `refcnt` of one.

Modify PTE of modifier to point to new page.

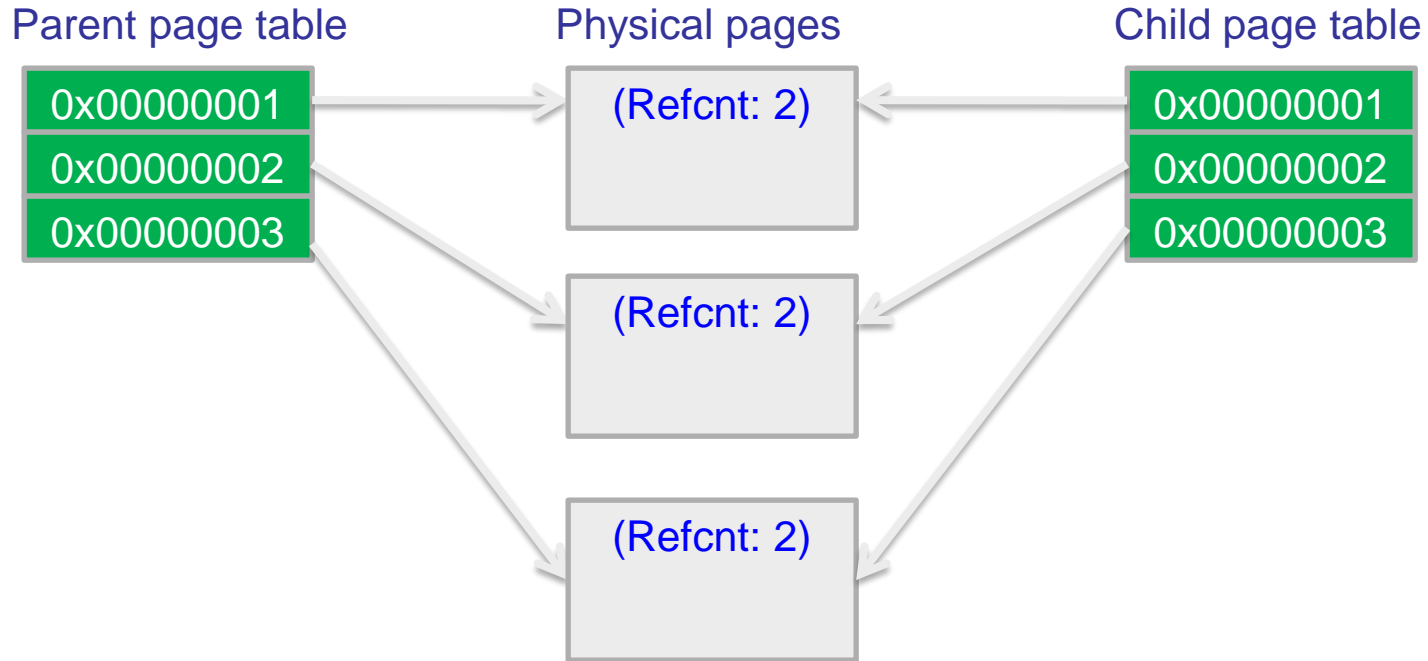
Decrement reference count of old page.

Copy-on-write: Example



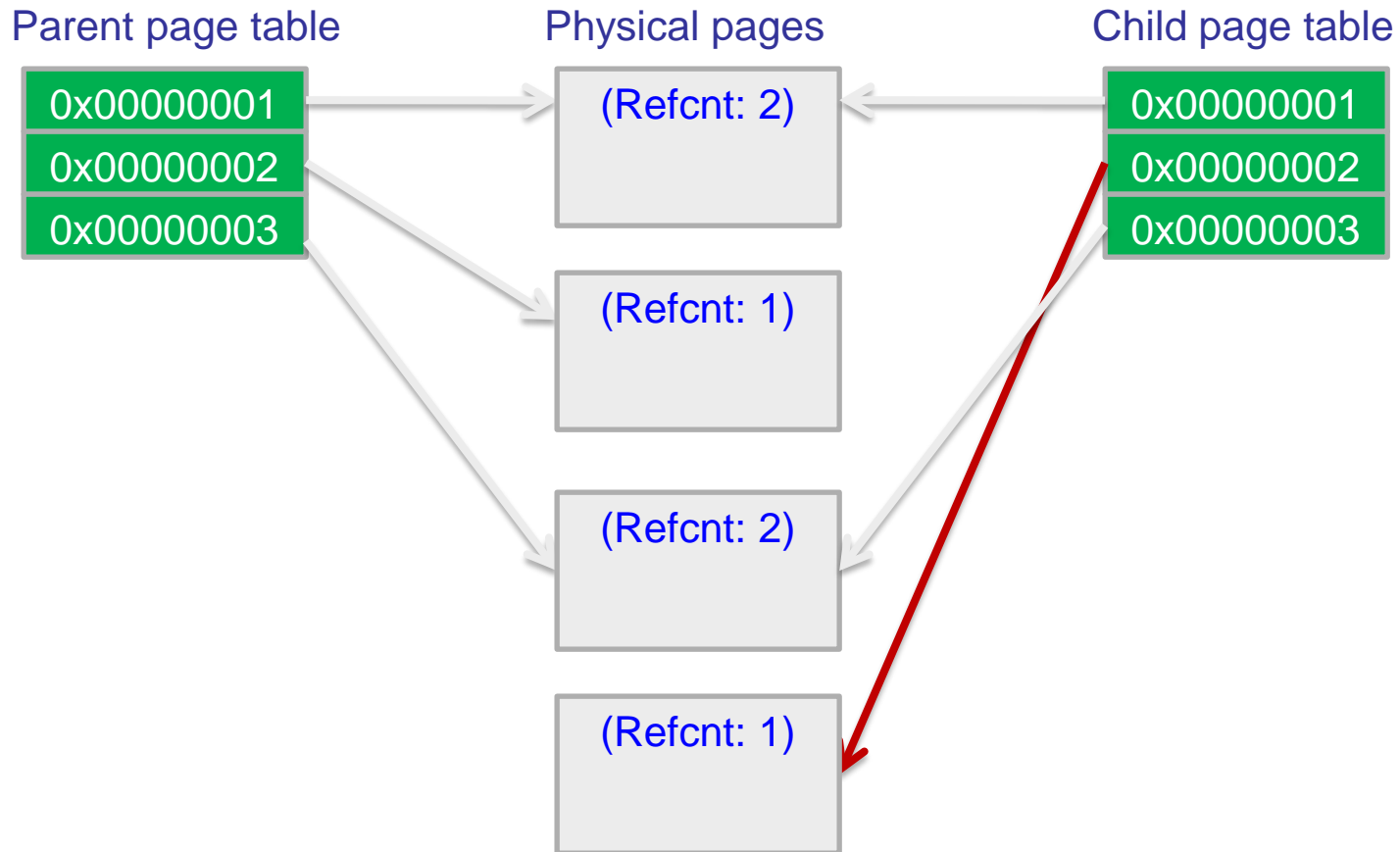
Parent about to fork().

Copy-on-write: Example



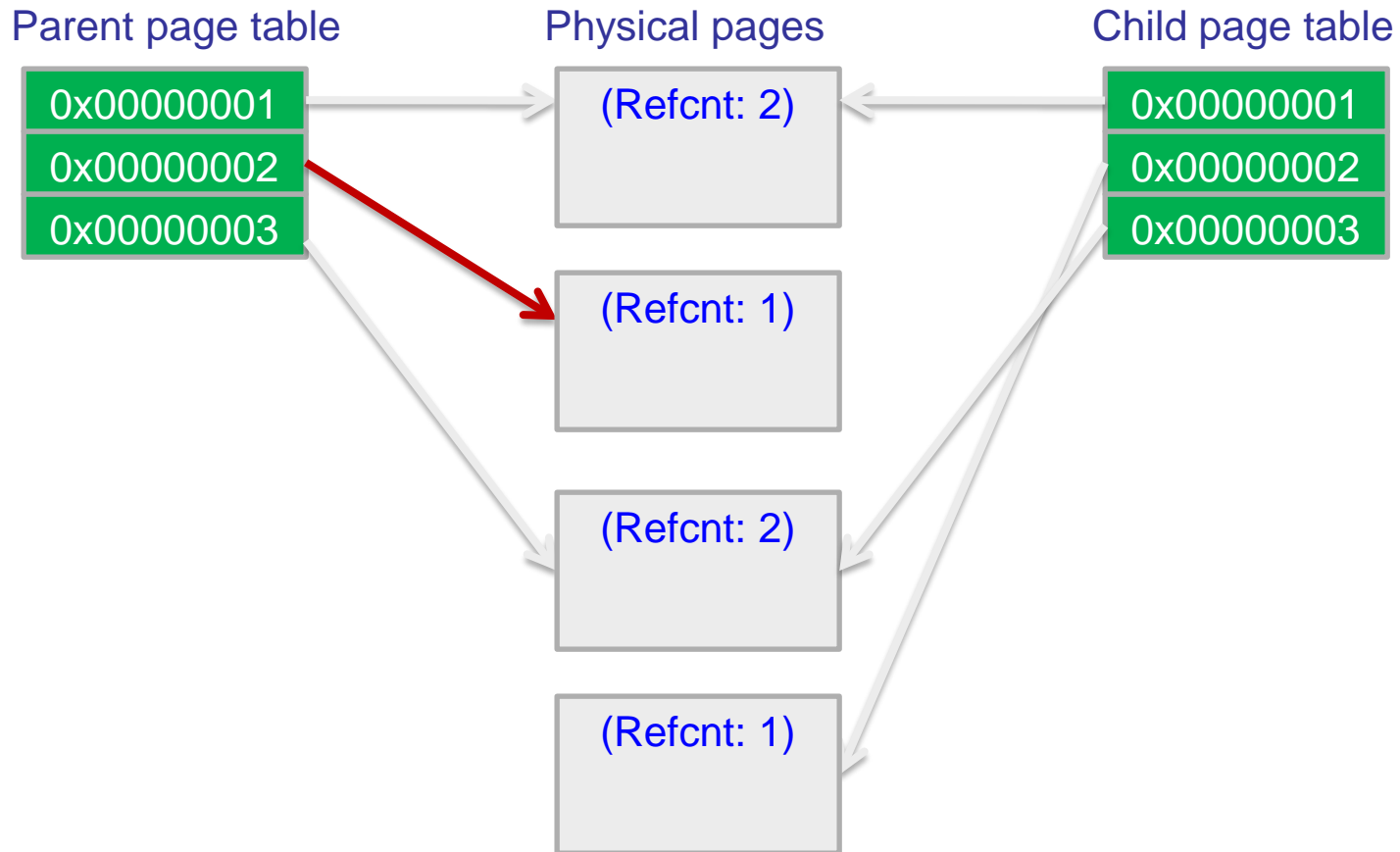
Copy-on-write of parent address space.

Copy-on-write: Example



Child modifying page 2 causes a copy to be made.

Copy-on-write: Example



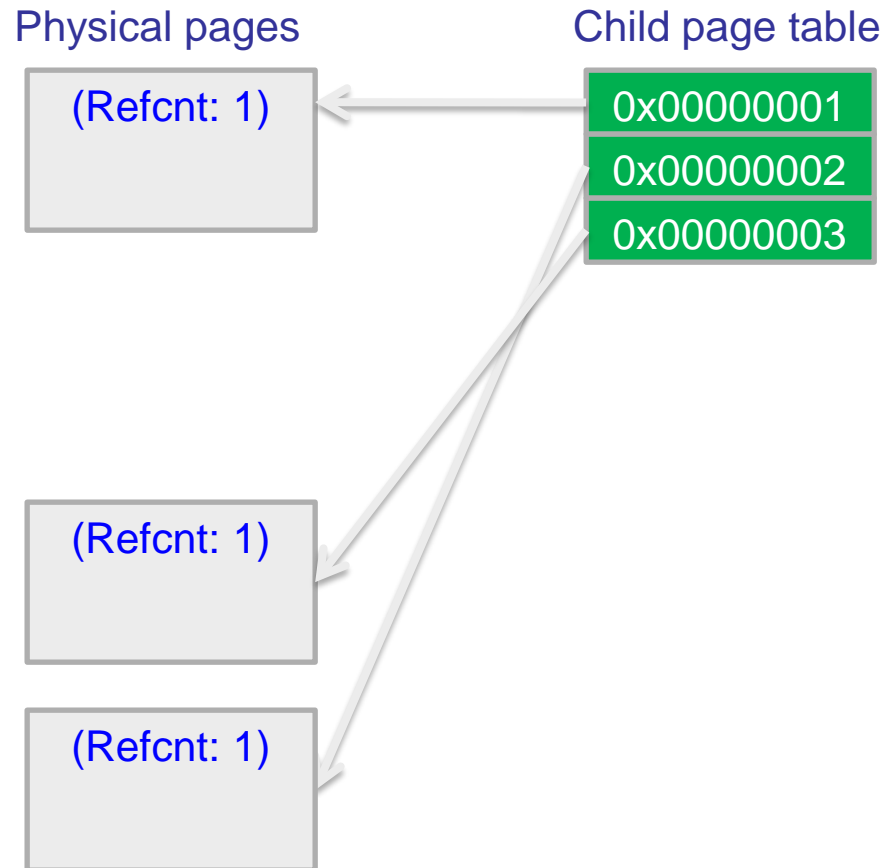
Parent modifying page 2 does not require copying.

Copy-on-write: Example

When the parent exits, its page table is deleted and the ref counts decremented.

If a ref count becomes 0, that page is freed.

The child may continue running.



Making exec() faster

exec() initializes code in the address space.

Naive solution: read file, copy into memory.

Can we do better?

Observation: most code never accessed.

Load code on-demand.

Similar to loading memory paged to disk.

Memory-mapped files, like the file-backed pages in P3.

File-backed vs. swap-backed

Swap-backed pages

Block on disk chosen by pager.

A process's writes to a page visible only to that process.

Modifications lost after process exit.

File-backed pages

Block on disk chosen by app.

Any process's write to a page visible to other processes that map the same block.

Modifications persist across process lifetimes.

Implementing a shell

Here's how a shell works.

```
while ( true )
{
    print the prompt;
    read the input;
    parse into a command + arguments;
    fork a copy of the shell process;
    if ( child )
    {
        redirect i/o as requested;
        exec the new program with the
            specified arguments;
    }
    else
        //parent
        if ( !background execution )
            wait for child to finish;
}
```

Agenda

1. Midterm.
2. Review of MLP and kernel vs. user.
3. Process creation and fork.
4. **Project 3 due July 27.**

Project 3

Process view:

1. Every process has an address space starting from `VM_ARENA_BASEADDR` of size `VM_ARENA_SIZE`.
2. When a process starts, the entire address space is invalid.
3. Process calls `vm_map` to make pages valid.
4. Pages becomes invalid when a process ends.

Pager view:

1. One process runs at a time.
2. Sets up page table that the MMU uses for translation.
3. Handles `vm_create`, `vm_map`, and `vm_fault`.

Project 3

Swap-backed pages:

1. Global swap file shared by all processes.
2. Pager controls where pages are stored in the swap file.
3. Individual pages are private to a process.

File-mapped pages:

1. Process specifies the file and offset.
2. Can be shared across processes.

Project 3: App vs. OS

Protection

All pages can be read from and written to.
Using R/W bits to track reference, dirty, etc.

Sharing

File-backed pages.
Copy-on-write.

Project 3

1. Do the project incrementally.
2. Swap-backed pages only without fork.
3. Then add support for fork and file-backed pages one after the other.
4. Pro Tip: Start with state diagrams for swap-backed, file-backed pages.

Project 3: State Diagram

For each unique state, consider:

1. Transitions? Read, write, clock, copy, ...
2. Attributes? Valid, resident, dirty, ...
3. Protections? Enable read, enable write?

