Midterm Review

Skadron, 2pm section, 10/9/2025

Getpid/waitpid/kill

- Fork creates a child process
  - Return value from fork in the parent is the new child's pid
    - Parent needs child's pid in order to call waitpid() — or maybe kill()
  - Return value from fork in the child is 0
    - To get its own pid, the child needs to call getpid()
- Waitpid allows parent to wait until specific child pid has completed and get its exit status
  - Exit status is what main returns or what a process provides as an argument to exit()
  - Exit status is an argument to waitpid – a pointer to an INT the parent has allocated, gets filled in by the kernel

Waitpid, etc - cont

- If parent calls waitpid before child exits, then the parent waits
- If parent calls waitpid after child exited, then the waitpid call can return immediately (although kernel may choose to context switch here)
- Waitpid with pid argument of -1 means wait for any child

- Parent can also kill a child using kill()
  - And then wait for it if desired
- Kill() can send any signal; by default it acts as CTRL-C, but it can also send SIGUSR signal, -9 (definitively kill), etc.
  - Kill = “send signal”
  - [This detail is not on exam]

Context switching

- Kernel may choose to context switch ANY TIME IT IS INVOKED
  - Whether through system call, exception, etc.

Exceptions

- Syscalls (a form of trap) – user software requests to invoke the kernel
- Interrupts - generally due to I/O events
  - Notifying the kernel something happened – disk read completed, user typed on keyboard, user pressed power button, etc.
  - An important example of an interrupt is a timer interrupt
- Faults – invoke kernel to handle error, eg divide by zero

- Each of these vectors to a handler by looking in the table entry that corresponds to the syscall/exception #
  - Get a function pointer for the appropriate handler

Syscalls, exceptions, etc.

- Things that invoke syscall - any invocation of OS services
  - Read, write, getpid, fork, exec, kill, etc.
- Syscall invokes a service on behalf of the user process
- Implementing that service may involve interrupts later
- Ex: user-level program does a read()

User level process does read()

- Two cases
  - 1. Input was received by kernel already
  - For example, user typed on keyboard
    - That generates interrupt, and kernel buffers that input for later
    - When user level process calls read() to get the keyboard input, that can return immediately (although OS might choose to context switch)
  - 2. Input not yet received by kernel
    - Process that calls read() will be suspended until desired input is ready
    - When user later types on keyboard, that causes interrupt, OS gets keyboard input and puts it in buffer, marks the requesting process as “runnable”, and then returns from read (at some point in the future)
- Note that with file I/O, you don’t have input magically appear before a read request

Open files across fork/exec

- Fork makes an exact copy of the parent, including open file descriptors
- So now parent and child share an open file instance
- So if parent and child are both writing to their respective fd's, their output will be intermingled (according to how they are scheduled)
  - Note here is that copies of fd's that point to the same open file will share the seek pointer
    - So writes from parents/child will be interleaved
- There is a global open file table, one entry per open() call, that contains state about the open file, including the seek pointer
  - Fd points to this open file table entry
- Every process has its own table of fd's, but inherited fd's are copies and thus point to whatever the original copy had
  - Child can change its fd without affecting parent, eg using dup2()
- Redirection should happen after fork and before exec
- Fd's are preserved across exec (this is the only part of the process that is preserved across exec; everything else is changed – program binary, stack, etc)

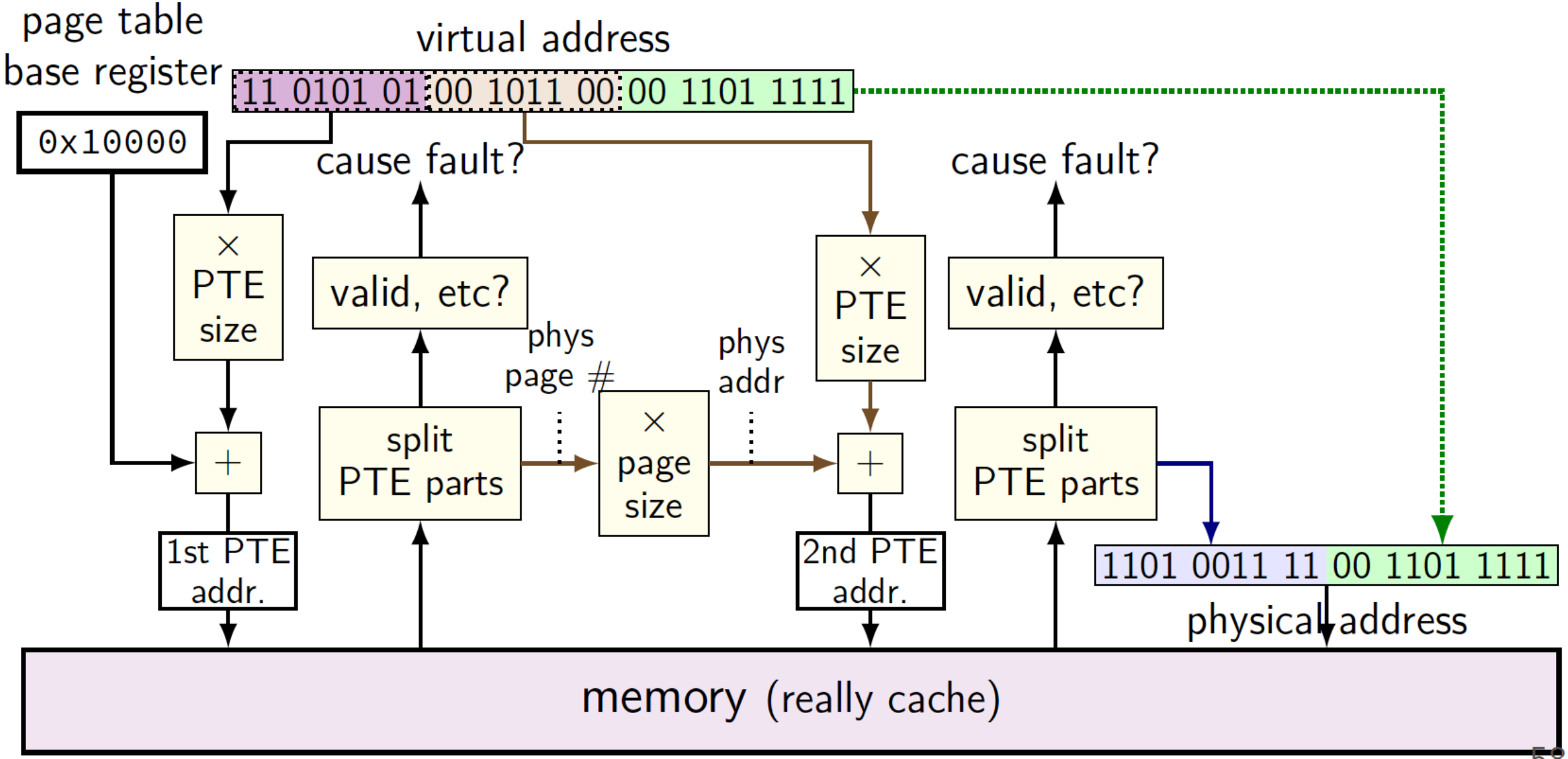
VA maps

- OS maintains a set of mapped VA ranges
  - Code, data, heap, stack, mmap’d files
- When process accesses a VA, if there is no valid PTE for that VA
  - Check map: if VA exists within mapped range, then create PTE
  - If VA does not exist within mapped range, then seg fault

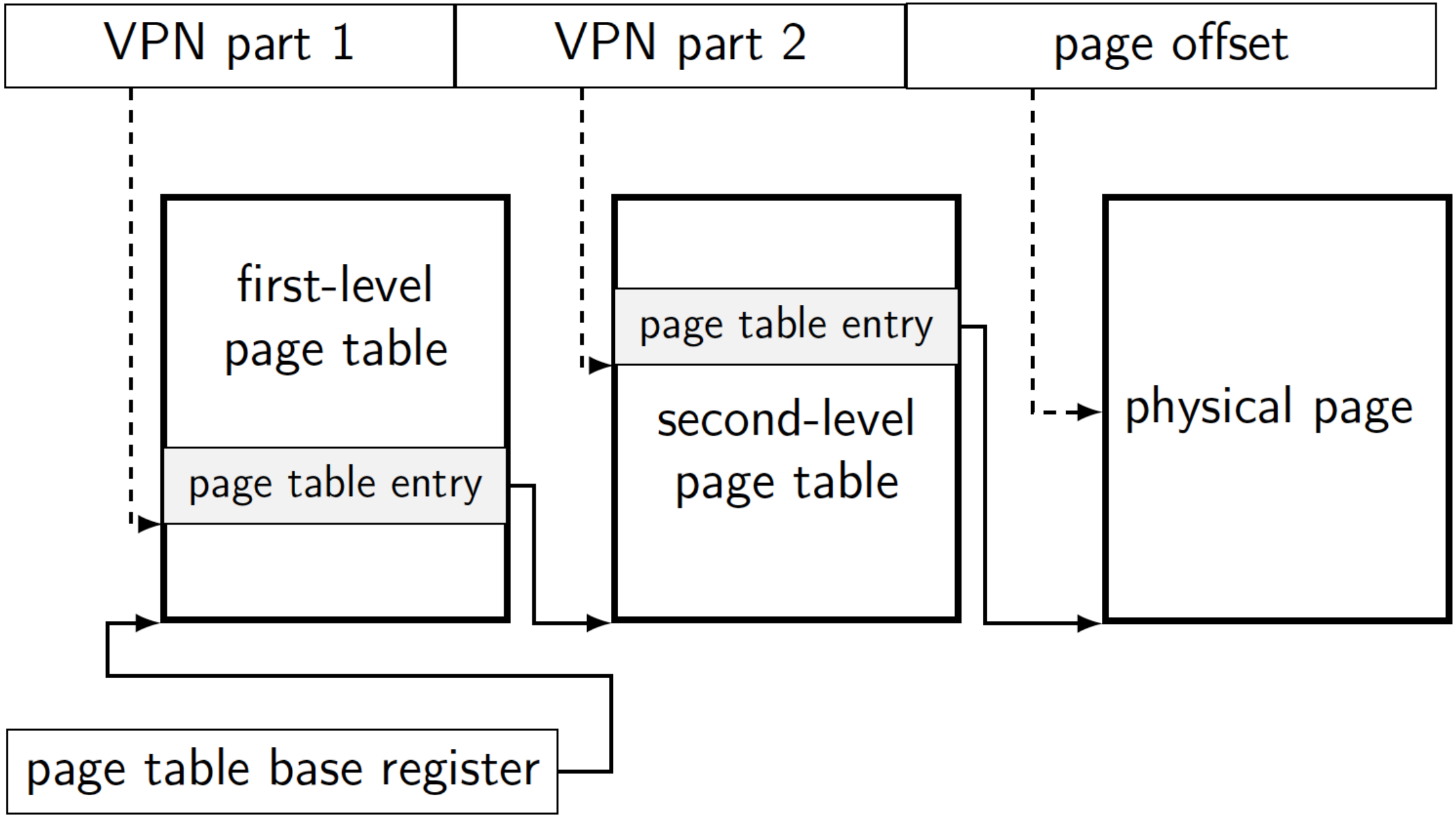
Translation tips

- Page offset is always lowest order bits in VA, # = log2 of page size
  - Eg, page offset for 4KB pages is lowest order 12 bits of VA
  - These are used directly in the PA as the lowest order bits
- VPN bits = VA size – offset bits
  - Eg, 32-bit VA, 4 KB pages ➔ 20-bit VPN
- For single-level translation, page table needs 2^VPNbits entries
  - Eg, in the example above, 2^20 entries
- For multi-level translation, if you know how big a specific page table is, and how big a PTE is, you know how many entries it has
  - Eg, if each page table is 4KB, and each PTE is 8 bytes, then you have 4 KB/8bytes =512 entries, and you will use log2(entries) bits from the VPN (9 in this example)
  - First level page table uses the highest order bits in the VPN, eg in the example above, you would use the high-order 9 bits to find the first-level page table entry
- To do the translation at a specific level:
  - PTE can be found at address = base address of this page table + (VPN part * sizeof PTE)
  - Then you take the contents of that PTE, break it into valid, user, etc. bits and the PPN
  - If there is another level of translation, multiply this PPN * sizeof(next level page table)
    - This will be the base address for the next level

Example translation flow – 2-level



another view



Other notes

- Table sizes don't have to be the same across levels
- We will always give you enough information to figure out the table size and/or how many VPN bits to use
- There is only one first-level page table (but it could be more than one page in size, although that would be an uncommon case)
- Each 1st-level PTE contains a PPN that points to a 2nd-level page table
  - # 2nd-level page tables = # entries in first-level page table