ASK: Wsparcie dla systemów operacyjnych

Krystian Bacławski

Główne zadania SO

- Wirtualizacja zasobów: czas, pamięć.
- Organizacja abstrakcji nad zasobami sprzętowymi: urządzenia I/O.
- Koordynacja współdzielenia zasobów sprzętowych.
- Izolacja procesów / zadań.
- Koordynacja procesów / zadań.

Pojęcia podstawowe

1. Przestrzeń adresowa.
2. Proces.
3. Program.
4. Jądro systemu operacyjnego.
5. System operacyjny.

Zegar

Współdzielenie czasu procesora – round-robin.
Kwant czasu, przełącznie kontekstu.

Czym jest kontekst? PC, rejestry, stos, etc.

Zalążek procesu – PCB.

Potrzebujemy mieć programowalne źródło przerwań o dobre rozdzielczości! (~1000Hz)

Izolacja

Program nie może zmienić stanu SO lub innych procesów w nieskoordynowany sposób!

- Potrzebujemy chronić stan procesu → ochrona pamięci.
- Nie możemy dać zagłodzić innych procesów lub systemu operacyjnego → wywłaszczanie.
- Musimy skoordynować dostęp do urządzeń I/O → tylko SO może komunikować się z urządzeniami I/O.

Wielozadaniowość a pamięć

Jak trzymać w pamięci wiele procesów?

- Gdzie załadować nowy proces? Relokacja?
- Gdzie każdy proces ma swój kod, dane, stos, stertę?
- Jak uchronić proces przed niepożądanymi działaniami innych procesów?
- Jak obsłużyć współdzielenie pamięci z innymi procesami?

Po co współdzielić zasoby?

- Oszczędność → biblioteki współdzielone.
- Komunikacja → współdzielone pliki / pamięć.

Czym innym jest współdzielenie od multipleksowania → dysk vs. pamięć.

Ten sam fragment pamięci musi być dostępny w obu procesach pod jakimś adresem (potencjalnie innym).

Izolacja

Co to jest przestrzeń adresowa?

Maszyna na najniższym poziomie operuje na adresach fizycznych → dostęp bezpośredni. Natomiast procesor posługuje się adresami efektywnymi.

Przestrzeń adresowa to funkcja częściowa “na”, stosowana przy każdym dostępie do pamięci!

$$M: ea \rightarrow \{phaddr, error\}$$

Rodzaje przestrzeni adresowych

Główne rodzaje przestrzeni adresowych:

- fizyczna → funkcja identycznościowa,
- logiczna → wszystko zmapowane na zasoby sprzętowe (pamięć operacyjna, urządzenia),
- wirtualna → pewne adresy nie są zmapowane, ale to nie błąd!

Błędy dostępu do pamięci

- bus error → błędny adres fizyczny
- protection violation → niewłaściwy dostęp
- page fault → brak mapowania
- unaligned access → niewyrównany dostęp

Procesor musi przekazać do SO informacje:

- adres instrukcji (`EPC`),
- adres dostępu (`BadVAddr`),
- rodzaj błędu (`Cause`).

Jedna globalna fizyczna PA

- Faktyczny stan, w którym startuje maszyna.
- Całkowity brak izolacji → wszystko jest współdzielone.
- Sami musimy się zatroszczyć o relokację → problemy z adresowaniem globalnym.

Bardzo przydatne relatywne tryby adresowania: względem rejestru lub licznika programu!

Jedna ciągła logiczna PA na proces

Procesor ma rejestr segmentu: `{phaddr, size}`. SO operuje w pamięci fizycznej.

```
M(ea) = ea >= size ? error : ea + phaddr
```

Przełączanie procesów → proste. Relokacja → zbędna! Współdzielenie i komunikacja → mocno ograniczone. Zmiana wielkości segmentu → utrudniona (fragmentacja). Nieefektywna wymiana → całymi segmentami.

Komunikacja z SO

Tylko SO może wykonywać pewne akcje → potrzebne wywołania systemowe (aka syscall).

Używamy przerwań programowych → instrukcje `int`, `trap`, `syscall` (często z #). Procesor przechodzi w tryb jądra i wykonuje *interrupt handler*. Co robi *handler*? Optymalizacja → alternatywny zestaw rejestrów do obsługi przerwania (`RDPGPR`, `WRPGPR`).

Tryb jądra / nadzorcy / użytkownika

Ograniczenie widocznej pamięci nie rozwiązuje problemów izolacji. Co jeśli program potrafi wyjść z *sandbox* używając instrukcji konfigurujących translację adresów?

Instrukcje uprzywilejowane → użytkownik nie ma uprawnień do konfiguracji procesora (`CP0`)!

Role trybów w architekturze MIPS.

Segmentacja a’la 80386 (1985)

Mamy tablicę deskryptorów segmentów: `[#num : {phaddr, size, flags}]`,

oraz kilka rejestrów segmentowych: kod (`cs`), dane (`ds`), stos (`ss`), inne (`es`, `fs`, `gs`). Flagi → `rwx`, `mode`, `present`, `accessed`. Rejestry segmentowe to indeksy tablicy deskryptorów. Modyfikacja tylko z poziomu SO.

`ea = seg:offset` (krotka!)

Segmentacja 80386 (c.d.)

Współdzielenie kodu segmentu → wiele instancji tego samego programu oszczędza pamięć! Współdzielenie segmentu `es` → IPC bez kopiowania! Bity `P` i `A` → wirtualizacja segmentów.

Izolacja działa już rozsądnie, ale zarządzanie pamięcią fizyczną i wymiana nadal bardzo uciążliwe!

Stronicowanie vs. segmentacja

Bardziej skomplikowana struktura opisująca PA.

```
ea = {vpn, offset}
```

Można konstruować PA z nieciągłych obszarów pamięci fizycznej! Możemy wymieniać pojedyncze strony!

To już było w “wykład11a.pdf”!

Tablica stron

Indeksowana numerem wirtualnej strony (`vpn`)

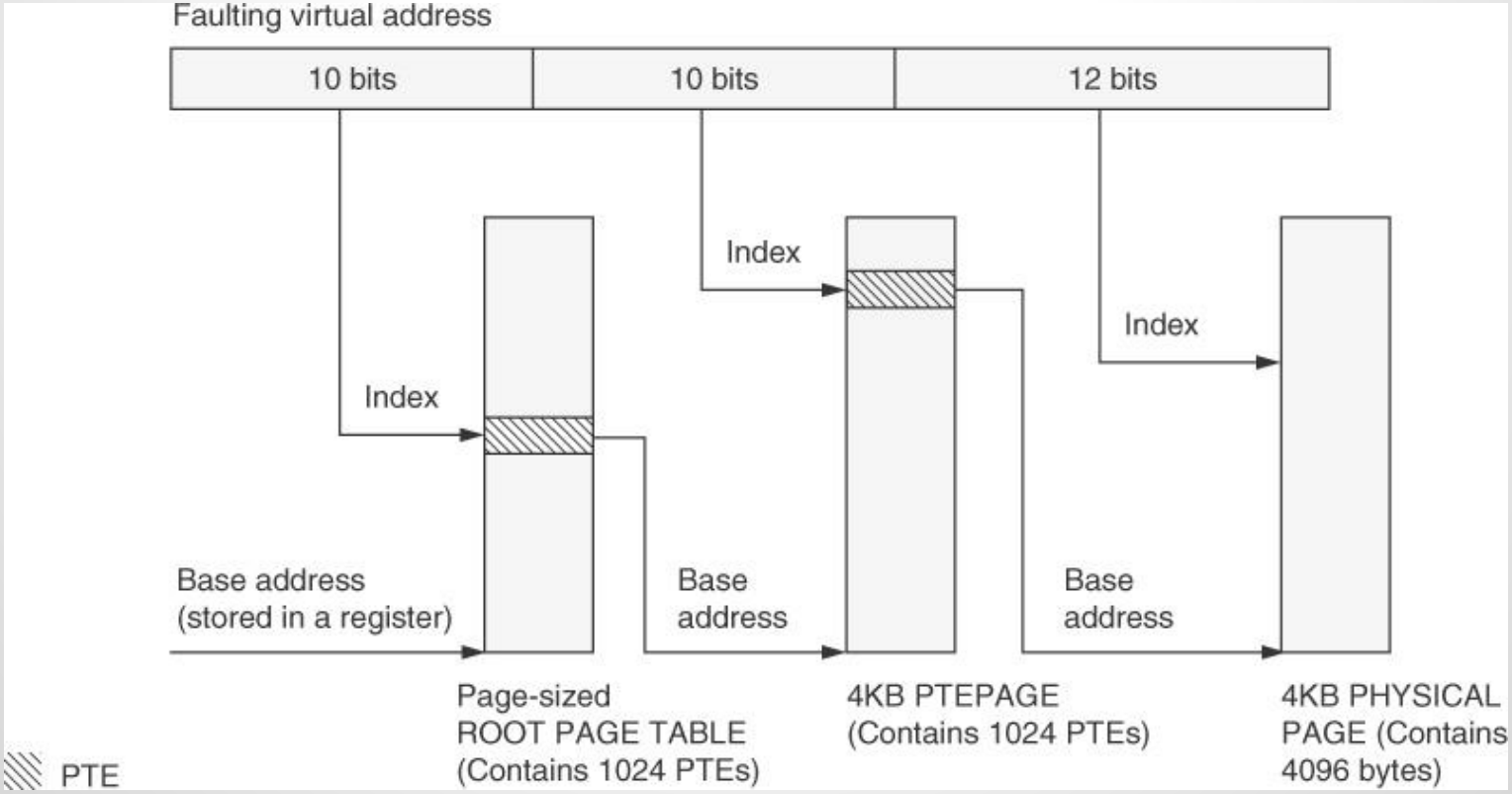
`[#vpn: {pfn, flags}]`

Gdzie flagi to `rwx`, `present`, `accessed`, `global`, `ASID`, `cache-control`. O trzech ostatnich później.

Wielkość tablicy stron? Niech PTE ma 4B, a strona 4096B. Dla 32-bitowej PA tablica stron ma 4MB! Nawet dla małego procesu!

Wielopoziomowa tablica stron

Oszczędzamy miejsce! Rozmiar rośnie liniowo wraz z ilością zmapowanych stron!



Translacja adresów w stronicowaniu

Dla `ea = {vpn, offset}` na każdy dostęp do pamięci mamy:

```
M(ea) = PT[vpn].pfn << K + offset
```

Załóżmy, że PT jest w fizycznej PA. Na każdy dostęp do pamięci wirtualnej robimy dwa dostępy do pamięci fizycznej?! A co przy wielopoziomowej tablicy stron?!

Musimy wprowadzić cache dla PTE → TLB!

TLB: przełączanie procesów

Przełączamy PA dwóch procesów. W procesie P1 strona S mapuje się na ramkę R1, a w procesie P2 na R2. Problem homonimów!

Musimy opróżnić TLB → boli! Albo dodać do każdej strony ASID → Address Space ID. Dodatkowy rejestr procesora → read-only dla użytkownika! Ograniczona liczba PA? Są lepsze rozwiązania → hybryda segmentacji i stronicowania w PowerPC.

TLB: współdzielenie pamięci

W P1 strona S1 i w P2 strona S2 mapują się na ramkę R. Przełączamy procesy – co wtedy?

Załóżmy, że S1 = S2 (np. fork, lub jądro Linux).
Brak ASID → może by zostawić mapowanie?
ASID → ale S1 i S2 są z innych PA!
Rozwiązanie: bit `global`!

S1 != S2 → problem z pamięcią podręczną!

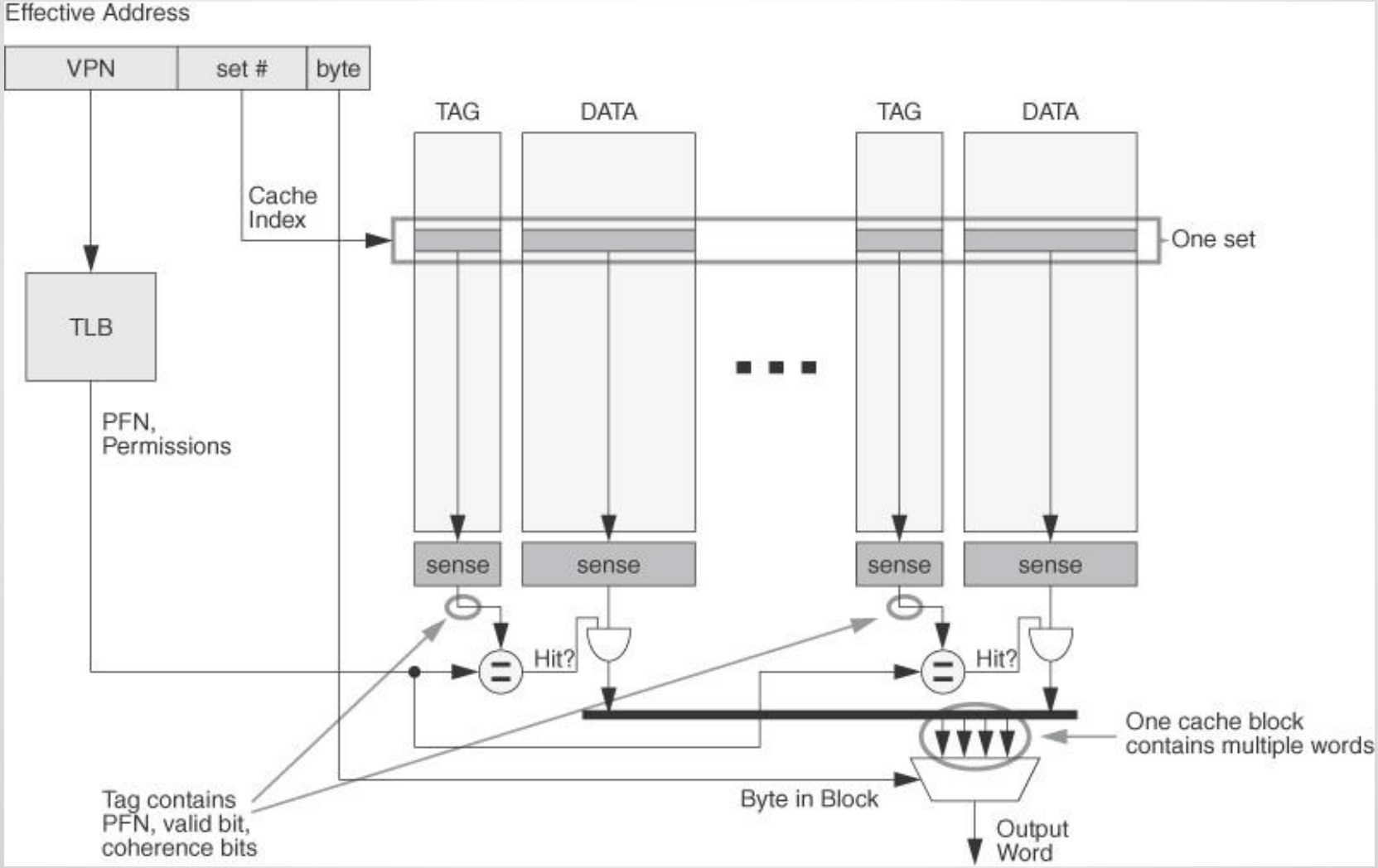
Translacja a pamięć podręczna

Żeby odnaleźć wiersz w pamięci podręcznej, wyciągamy indeks z adresu – którego? Porównujemy znacznik przyczepiony do wiersza – z którym adresem?

Najpopularniejsze rozwiązanie: virtually-indexed, physically-tagged. Czemu?

Jednocześnie sprawdzamy czy strona jest zmapowana (znacznik + uprawnienia dostępu) i wyszukujemy w pamięci podręcznej.

Virtually-indexed, physically-tagged, n-way set associative cache

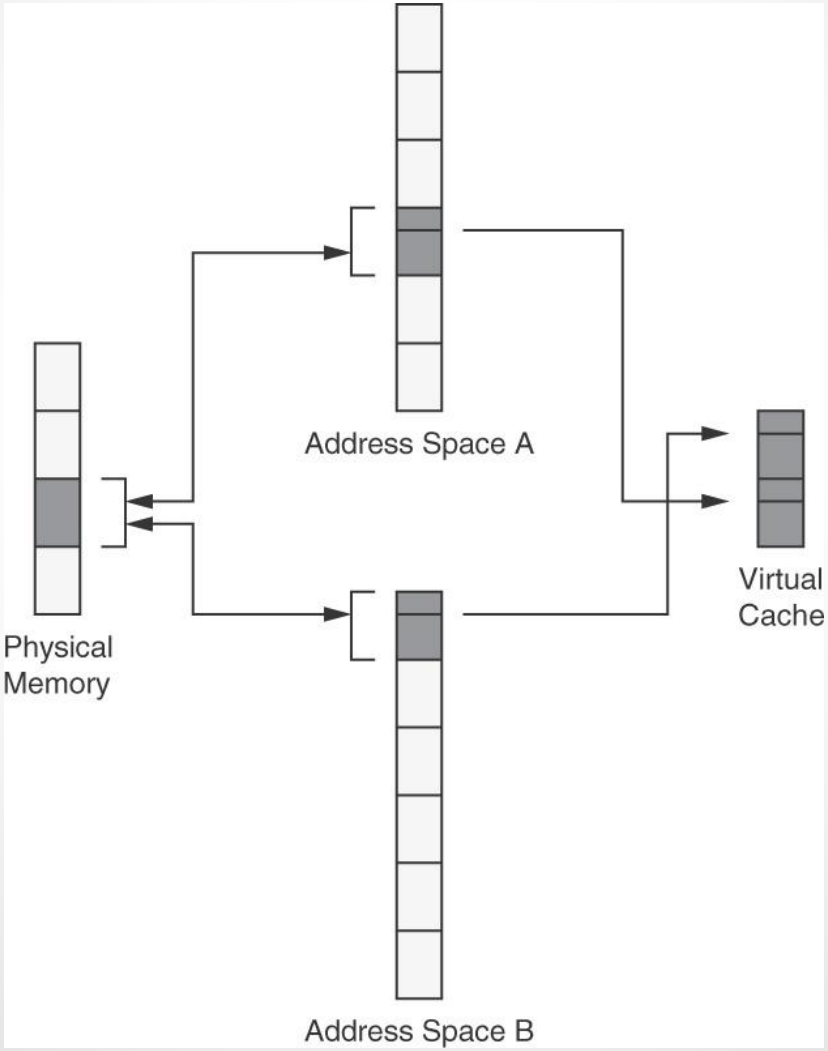


Problem synonimów

Szybko, ale mamy problem kiedy współdzielimy pamięć. P1: S1 → R, P2: S2 → R, S1 != S2.

Powstają tzw. synonimy! Możemy mieć zdublowane linie w pamięci podręcznej... z różnymi danymi! Musimy wyczyścić pamięć podręczną przy przełączaniu procesów → bardzo boli! Ew. triki z obłożeniem PA.

Problem synonimów (c.d.)



Polityka zapisu

Rysujemy grafikę na ekran → zapisujemy do pamięci karty graficznej! Co jeśli `write-back`?

Część danych mogła zostać w pamięci podręcznej i karta graficzna wyświetli nam niekompletny obraz!

Do PTE dodajemy bity kontrolne, które mówią jak ma się zachowywać pamięć podręczna dla adresów z tej strony!

Ustawmy `write-through`!

Rejestry urządzeń a cache

Rejestry urządzeń zmapowane w fizycznej PA. Zmieniają się (np. temperatura, ilość przesłanych bajtów), ale procesor ma w pamięci podręcznej ich stare wersje!

Ręcznie unieważnianie bloków → instrukcja uprzywilejowana, możemy zapomnieć!

Dodajmy flagę w PTE → `non-cacheable`.

Minimalizacja chybień TLB (x86)

Chybienie TLB bardzo kosztowne! Dla wielopoziomowych tablic stron → koszmar!

Zdarza się często dla dużych nieliniowo ułożonych struktur danych → głębokie drzewa.

Dla wielopoziomowej tablicy stron wprowadźmy duże strony – np. $2^{(12+10)}$B = 4MiB. Potrzebne specjalne API w SO.

Zarządzenie pamięcią w architekturze MIPS

Stronicowanie

Nie ma sprzętowego mechanizmu wypełniania TLB → TLB miss exception.

`CP0`: `Index`, `Random`, `EntryHi`, `EntryLo`, `PageMask`. Instrukcje uprzywilejowane:

TLBP → odnajduje wpis TLB odpowiadający adresowi w rejestrze `Index`.

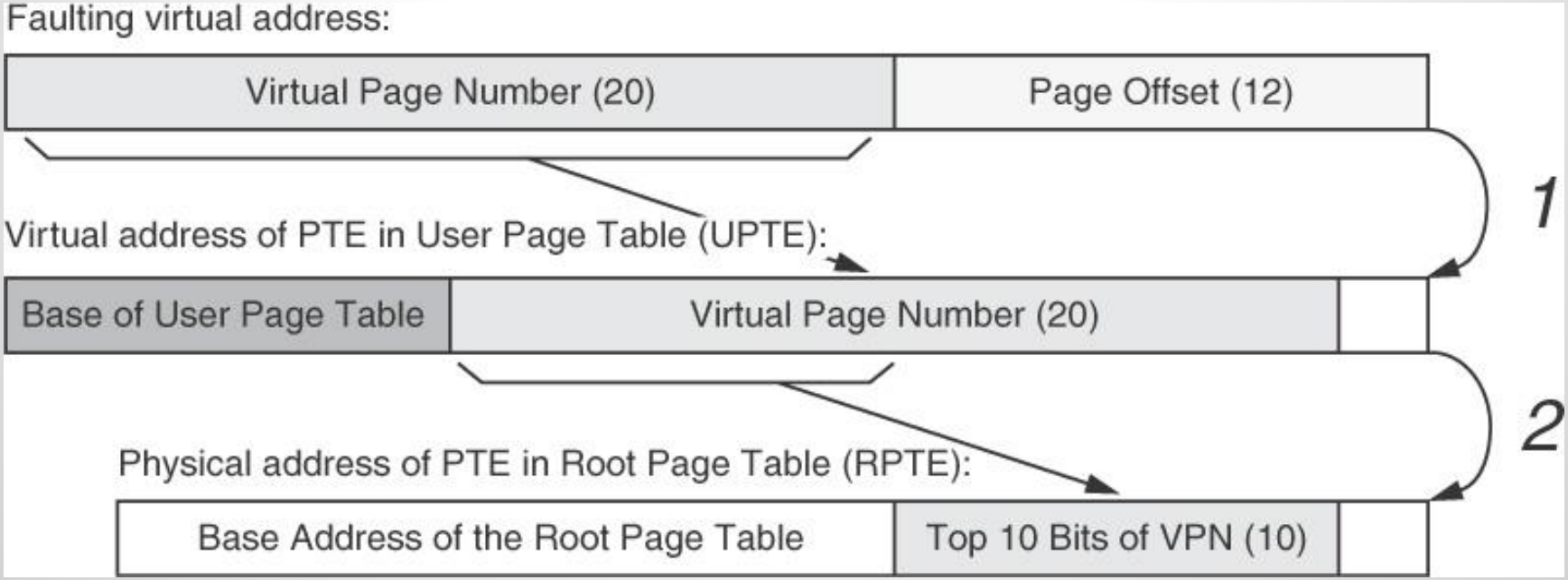
TLBR → odczytuje daną pozycję TLB,

TLBWI → zapisuje PTE pod dany indeks TLB,

TLBWR → zapisuje PTE pod losowy indeks.

Organizacja tablicy stron

UPTE w wirtualnej PA jądra. RPTE w fizycznej PA. Metoda przeglądania `bottom-up`. Strony mogą być różnej wielkości → można mieszać.



Prefetching

```
PREF hint, offset(base)
```

Podpowiadanie procesorowi jak ma korzystać z pamięci podręcznej. Intencje programisty:

- L/S → mam zamiar czytać / pisać
- streamed L/S → iteracja po tablicy,
- retained L/S → wielokrotne przeszukiwanie drzewa (wierzchołki blisko korzenia),
- WB invalidate → niepotrzebne, wyrzuć.

Kod samomodyfikujący się

L1d i L1i są rozłączne → wydajność i brak hazardów! Dane w L1i tylko się zastępuje.

Piszemy kompilator JIT (jak w zadaniu z BF). Nadpisujemy ciąg instrukcji → modyfikacja L1d. L1i pozostaje niezmienione. Co zrobić?

Unieważnić linie w L1i → nieuprzywilejowana instrukcja `SYNCI offset(reg)`!

Synchronizacja

Wyścig (*race condition*)

Co jeśli poniższy kod wykonują dwa procesy, a przełączanie jest niedeterministyczne?

```
1: lw   $t1, ($t0)
2: addi $t1, $t1, 1
3: sw   $t1, ($t0)
```

Może się zdarzyć, że dodawanie wykona się dwa razy, ale wartość komórki zwiększy się o 1.

Rozwiązanie brzydkie

Instnieją rozwiązania algorytmiczne → powolne, skomplikowane, dużo założeń.

Maszyna jednoprocesorowa → użyjmy instrukcji `DI` / `EI` (Disable / Enable Interrupt).

- `DI` i `EI` → instrukcje uprzywilejowane
- możemy przyblokować b. ważne przerwanie!
- błędny program blokuje działanie SO!

Porównaj i zamień – CAS

Instrukcję atomowa → nie widać pośrednich efektów wykonania.

```
int CAS(int* reg, int oldval, int newval) {
  atomic {
    int old_reg_val = *reg;
    if (old_reg_val == oldval)
      *reg = newval;
  }
  return old_reg_val;
}
```

Sekcja krytyczna i CAS

```
1: loop:
2:    lw   $t1, ($t0)
3:    addi $t2, $t1, 1
4:    cas  $t3, ($t0), $t1, $t2
5:    bne  $t3, $t1, loop
```

CAS nie da się zakodować w architekturze MIPS. Wymaga dwóch etapów MEM! Rozmiar określony z góry! Nie jest odporny na problem ABA → podmiana walizki na lotnisku.

Pamięć transakcyjna

1. Kopiujemy atomicznie zawartość bloku.
2. Modyfikujemy naszą kopię wedle uznania.
3. Jeśli oryginał nie uległ modyfikacji w międzyczasie → zapisujemy atomicznie naszą kopię do oryginału.

- Wykrywamy ABA!
- Jaka jest dopuszczalna wielkość bloku?
- Ile takich niezależnych bloków na raz?

Load-linked / store-conditional

Ograniczona implementacja sprzętowa bazująca na blokach pamięci podręcznej:

```
blok = {tag, dane, flagi}
```

Bazujemy na fladze *modified*. Śledzimy listę bloków oznaczonych przez LL. Jeśli zostały zmodyfikowane to SC nie zapisze i zwróci błąd.

Dobre do synchronizacji na maszynach wieloprocesorowych! Na pojedynczym procesorze trzeba uważać!

LL / SC w architekturze MIPS

Tylko jedna otwarta transakcja na procesor.

```
1: loop:
2:    ll   $t1, ($t0)
3:    addi $t1, $t1, 1
4:    sc   $t1, ($t0)
5:    beqz $t1, loop
```

Udana atomiczna sekwencja RMW → 1, w p.p. 0. `LL` ustawia `LLBit`, `PAUSE` oczekuje na 0.

Bariery pamięciowe

Out-of-order → kolejność wykonania instrukcji jest wyznaczana w trakcie działania przez procesor. Również operacji na pamięci!

```
1: lw   %t0, ($a0)  # $a0 → licznik
2: addi %t0, %t0, 1
3: sw   %t0, ($a0)
4: sw   $0, ($a1)   # $a1 → blokada
```

Co jeśli procesor przestawi zapisy w 3 i 4 ?

Bariery pamięciowe (c.d.)

Instrukcja realizująca barierę pamięciową → wymuszenie kolejności odczytów / zapisów.

```
SYNC stype
```

Które z wcześniejszych instrukcji mają się zakończyć przed SYNC? Które z późniejszych operacji muszą się zacząć po SYNC?

Inne

Debugowanie

Asercje:

- `BREAK`
- Pułapki: `TEQ` / `TGE` / `TLT` / `TNE`.

Debugowanie krokowe:

- tryb debugowania (`DERET`),
- breakpoint → `SDBBP`,
- powrót z debuggera.

Profilowanie

Dostęp do liczników sprzętowych:

- cykle procesora (`RDHWR`),
- cache L1 / L2 hit / miss,
- TLB hit / miss,
- błędnie przewidziane skoki,
- ilość słów przesłanych między CPU i RAM,
- etc.