Betriebssysteme Übung
Tutorium „TLB & Virtual Memory“

Task 1 – TLB (1)

bisher:

- jeder Prozess hat seine Pagetable
- CPU hat Zeiger auf die aktuelle Pagetable (des gerade laufenden Prozesses) im Speicher

Nachteil:

- Für jede Adressübersetzung ist ein Speicherzugriff erforderlich
- z.B.. Intel Celeron 1GHz: Front Side Bus läuft mit 100MHz; um **Faktor 10 langsamer**!

Task 1 – TLB (2)

Lösung: Cache auf die Pagetable

- **TLB** (Translation Lookaside Buffer)
- assoziativer Speicher: paralleler Vergleich aller Einträge
- Hardware!
- Größe begrenzt (4-32 Einträge)
- enthält die selbe Datenstruktur wie die Pagetable
- Update entweder selbständig durch die CPU oder durch das **Betriebssystem**

Task1 – TLB (3)

in Nachos:

- Compilerflag `-DUSE_TLB`

  ab sofort verwendet Nachos den TLB
- Pagetable in der Maschine nicht mehr zuweisen

  (machine->pageTable = currentThread->space **entfernen**)

  da *entweder* PT *oder* TLB verwendet wird

Task 1 – TLB (4)

Ablauf der Adressübersetzung:

- virtuelle Adresse im TLB suchen
- gefunden: physikalische Adresse aus TLB verwenden
- nicht gefunden: **PagefaultException**
- ExceptionHandler wird aufgerufen und muss den TLB entsprechend aktualisieren
- gesuchte virtuelle Adresse im `BadVAddrReg`-Register

Task 1 – TLB (5)

TLB-Replacementstrategien (z.B.)

- FIFO

  nicht besonders gut, da auch häufig benötigte Einträge ersetzt werden (nicht implementieren!)
- Clock
- 2nd Chance
- ...

Bonus: mehrere implementieren und Performance (Anzahl der Pagefaults) vergleichen

Task 1 – TLB (6)

Achtung:

- Taskswitch: SaveState & RestoreState, TLB löschen oder (besser) speichern und wiederherstellen
- SC_EXEC: TLB initialisieren
- in SystemCalls: ReadMem und WriteMem könnten Pagefault auslösen! Exceptionhandler reentrant oder vor jedem Speicherzugriff „zu Fuss“ den TLB aktualisieren

Task 2 – Virtual Memory (1)

bisher: genügend (unendlich) großer Hauptspeicher

ab sofort:

- `#define NumPhysPages 16`
- Erzeugen einer Auslagerungsdatei
- `pageTable[i].valid = FALSE;` wenn die Seite nicht im RAM steht
- Hauptspeicher ist Cache für das Swapfile

Task 2 – Virtual Memory (2)

mögliche Ansätze:

- Prepaging:
  - Speicher für Prozess wird vorreserviert
  - rel. einfach zu implementieren
- Demandpaging
  - beim Prozessstart wird noch kein Speicher reserviert
  - `pageTable[i].physicalPage = NOTLOADED;`

    wenn die Seite noch nie geladen wurde
  - rel. aufwändig zu implementieren

Task 2 – Virtual Memory (3)

Datenstrukturen:

- Bitmap für Swapfile
- inverted PageTable

  Verknüpfung zwischen Pages im Swapfile und Prozessen
- PageTable.valid, .dirty, .used sind interessant für Swapping-Strategie
- PageTable.physicalPage kann beim Auslagern als Adresse im Swap verwendet werden, sofern ein Algorithmus ohne Kopien verwendet wird

Task 2 – Virtual Memory (4)

Swapping-Replacementstrategie

- Wieder: FIFO ist suboptimal (nicht implementieren)
- es können die selben Algorithmen wie beim TLB-Replacement angewandt werden
- Variante: Beim Swap-In Kopie im Swapfile belassen, dann muss nicht ausgelagert werden, wenn die Seite nicht verändert wird;
  allerdings wird dann eine eigene „PageTable“ für das Swapfile benötigt (PT-Einträge sind Hardwareabhängig und dürfen nicht erweitert werden!)

optionaler Task 3 – Checkpointing

- neuer Systemcall
- ein Prozess wird in seinem momentanen Zustand „eingefrohren“ und in ein File ausgelagert
- anschließend wird er komplett aus dem VM entfernt
- geöffnete Dateien müssen nicht beachtet werden
- Fortsetzen des mit SC_EXEC
- Denkanstoss: Task-Switch (auch dabei muss der Prozessstatus möglichst vollständig gespeichert werden)