Výpočet v módu jádro

v důsledku událostí

- přerušení (od zařízení asynchronně)
- výjimky
- softvérové přerušení

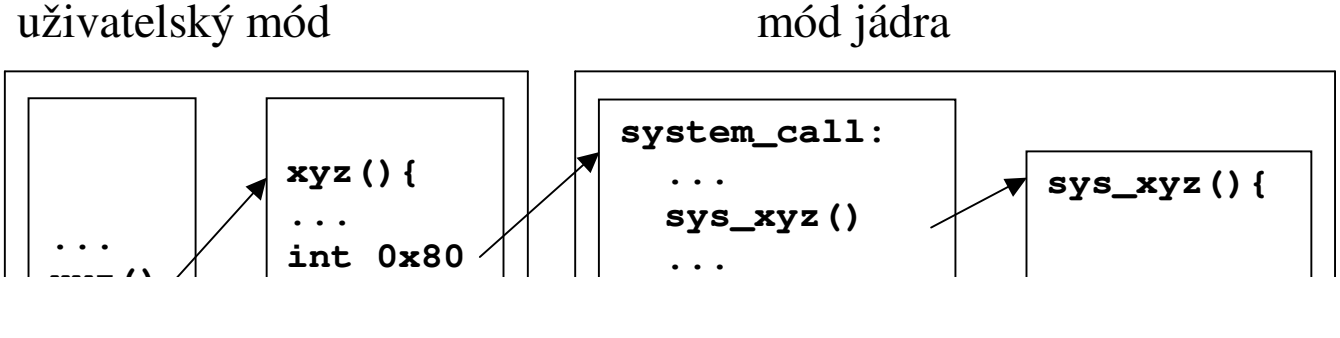
řízení se předá na proceduru pro ošetření odpovídající události

část stavu přerušeného procesu potřebná pro obnovení jeho vykonávání po skončení obsloužení události (počítadlo instrukcí, PSW) se uloží do zásobníku jádra přerušeného procesu

Sytémové volání

v standardní knihovně jazyka C je pro každé systémové volání obálková procedura, řízení se předá softvérovým přerušením proceduře jádra, která se nazývá **`syscall( ),`** **`system_call,`** protože je jedna pro všechny služby, požadovaná služba je identifikována parametrem procedury, který se nazývá číslo systémového volání

Linux



aplikační program volá službu **`xyz`** (např. **`fork()`**)

obálková procedura uloží číslo služby do registru **`eax`**
(5 pro **`fork()`**) před vykonáním **`int 0x080`**

`system_call:`

- uloží obsah registrů (HW kontext)
- zavolá odpovídající funkci (v jazyku C)
- ukončí se voláním **`ret_from_sys_call()`**

Výjimky se spracují obdobně

jsou synchronní s procesem (vznikají v důsledku událostí způsobených vykonáváním procesu)

procedury pro jejich zpracování mají obdobnou strukturu jako procedura pro systémová volání **`system_call`**
Linux

Procedura pro zpracování výjimky:

- uloží obsah registrů
- zpracuje výjimku (funkce v jazyku C)
    - pošle signál procesu
    - zpracuje žádost o stránku
- ukončí se voláním funkce **`ret_from_exception()`**

Zpracování přerušení

přerušení je obecně asynchronní vzhledem k přerušenému procesu

- proces čeká na přenos dat, po dokončení přenosu je přerušen úplně jiný proces

zpracování přerušení nesmí způsobit čekání, přerušený proces zůstává ve stavu běžící

čas zpracování přerušení je započítán přerušenému procesu, při zpracování přerušení se tedy přistupuje do jeho záznamu **`proc`**

obsloužení přerušení:

- uloží IRQ (*Interrupt ReQuest)* a obsah registrů
- pošle potvrzení PIC (*Programmable Interrupt Controller)*
- vykoná obslužní proceduru přerušení
- ukončí se skokem na **`ret_from_intr()`**

Vzájemné vnoření systémového volání, výjimek a přerušení

Předpokládejme odladěné jádro

1. zpracování systémového volání
   - může vzniknout výjimka žádost o stránku (výpadek stránky)
   - může vzniknout přerušení

2. zpracování výjimky (jakékoliv)
   - může vzniknout přerušení

3. zpracování přerušení
   - může vzniknout přerušení

při každém odkladu zpracování některé z uvedených událostí musíme uložit odpovídající HW kontext

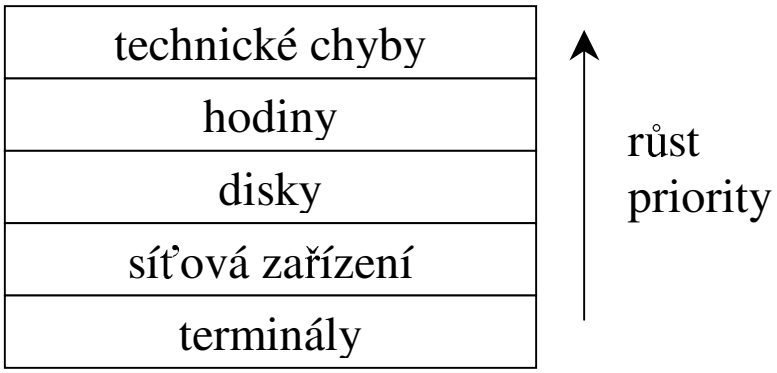
- vytváří se kontextové vrstvy v zásobníku jádra přerušeného procesu
- existuje globální zásobník přerušení

přerušení nejsou všechna stejně naléhavá

- prioritní schéma
- odložení vykonání nekritických akcí obsluhy

prioritní schéma

- přerušení mají přiřazené prioritní úrovně (*interrupt priority level*)

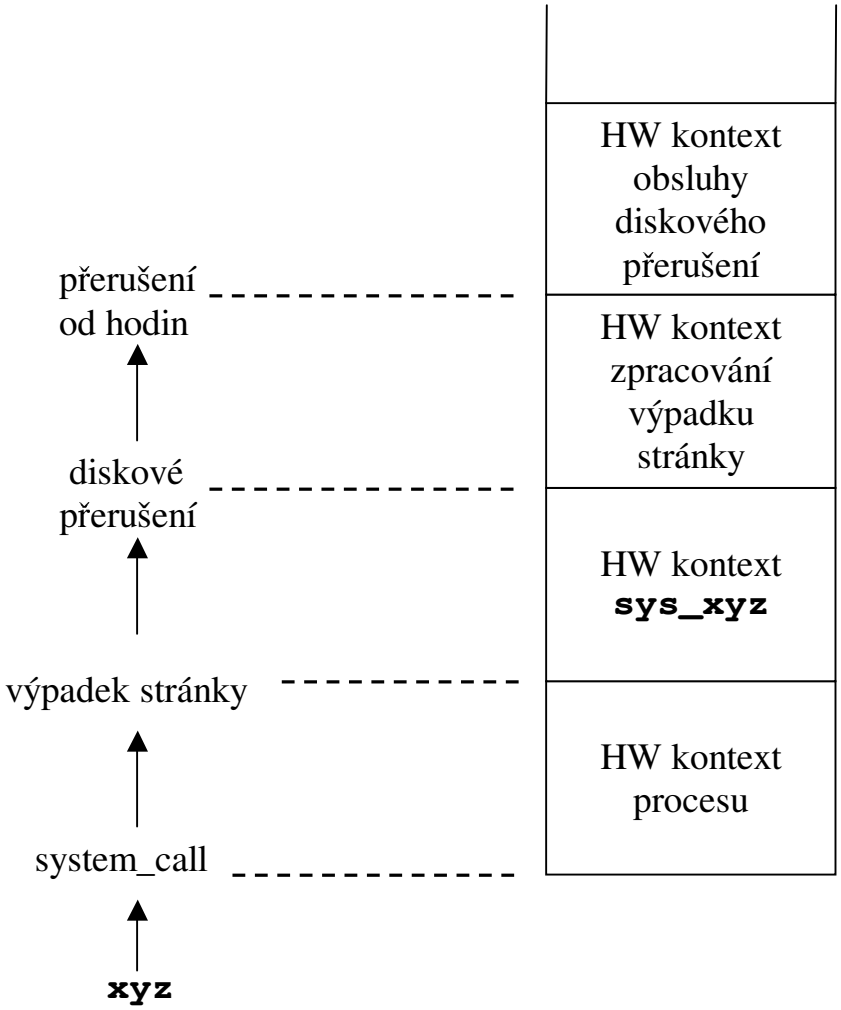


- ve stavovém registru procesoru je nastavena okamžitá prioritní úroveň zpracovávaného přerušení
- vznikne-li přerušení s nižší nebo stejnou prioritní úrovní, je uloženo a jeho obsluha je odložena
- vznikne-li přerušení s vyšší prioritní úrovní, uloží se HW kontext, na tuto vyšší prioritní úroveň se nastaví hodnota okamžitého přerušení ve stavovém registru procesoru, zpracuje se přerušení
- při skončení zpracování se z uloženého PSW obnoví okamžitá prioritní úroveň přerušení

Příklad:

aplikační program → **`xyz`** → výpadek stránky →
přerušení od terminálu → diskové přerušení →

přerušení od hodin



odložení vykonání nekritických částí obsluhy přerušení (Linux)

akce při obsluze přerušení se dělí do tří tříd

- kritické, potvrzení přerušení, aktualizace dat používaných zařízením i procesorem, přerušení jsou zakázána (*disabled*)

- nekritické, data používána jenom procesorem, přerušení uvolněna (*enabled*)

- nekritické odložitelné, kopírování vyrovnávací paměti do adresového prostoru procesu, vykonávané funkcí jádra *bottom half*, vykonají se však před **`ret_from_intr()`**

umožňuje prokládání (*interleaving*) vykonávání obsluh přerušení

- zlepšuje propustnost PIC a řadičů zařízení, nemusí čekat na dokončení obsluhy předcházejícího přerušení
- zjednodušuje kód jádra a zlepšuje přenositelnost

Vytvoření procesu – `fork()`

1. Přiděl nový PID a **`proc`** záznam.
2. Inicializuj **`proc`** záznam potomka.
   - UID a GID, maska signálů, ... se kopírují z rodiče

- čas využití CPU, ... se nulují
- PID, PID rodiče se nastaví pro potomka.

3. Zvyš počet odkazů na i-uzel okamžitého adresáře a případně na i-uzel změněného kořenového adresáře...
4. Zvyš počet odkazů na otevřené soubory v tabulce souborů.
5. Přiděl potomkovi tabulku oblastí.
6. Přiděl potomkovi u oblast a zkopíruj ji z rodiče.
7. Přidej potomka k procesům sdílejícím oblast textu (kódu), který vykonává rodič.
8. Zdvoj oblast dat a zásobníku.
9. Inicializuj HW kontext potomka kopírováním registrů rodiče.
10. Nastav stav na **`připraven na vykonání`** a vlož ho do fronty pro plánovač.
11. Potomkovi vrať hodnotu 0.
12. Rodiči vrať hodnotu PID potomka.

optimalizace

`fork` vždycky vytvářel nový adresový prostor pro potomka

- *copy-on-write* (kopíruj při zápisu), System V a další
  - potomek dostane vlastní kopii tabulky oblastí
  - oblast dat a zásobníku (jejich stránky) jsou dočasně í *read-only* (přístup jenom pro čtení) a označené jako *copy-on-write*
  - pokusí-li se rodič nebo potomek modifikovat stránku těchto oblastí vznikne výjimka
  - výjimka se zpracuje tak, že se vytvoří zapisovatelná kopie stránky

- zavolá-li potomek **exec** nebo **exit**, stránkám rodiče se vrátí jejich původní ochrana

- systémové volání **vfork()**, BSD
  - jestli očekáváme po **fork** brzké volání **exec** můžeme použít nové systémové volání **vfork()**
  - potomek je vykonáván v původním adresovém prostoru rodiče, který spí do jejich vrácení
  - když potomek vykoná **exec** nebo **exit** jádro adresový prostor rodiči a vzbudí ho

Vyvolání programu – **exec()**

program je ve vykonatelném souboru – **a.out, coff, elf**

1. Analyzuj cestu k souboru a zpřístupni soubor, i-uzel.
2. Ověř, že volající má oprávnění na jeho na jeho vykonání.
3. Přečti hlavičku souboru a ověř, že soubor je vykonatelný.
4. Má-li soubor nastaven bit SUID nebo SGID, změň efektivní UID a uložený nastavovací UID na UID vlastníka souboru.
5. Zkopíruj argumenty a proměnné okolí do adresového prostoru jádra.
6. Uvolni paměťové oblasti procesu. Byl-li proces vytvořen službou **vfork**, vrať adresový prostor rodiči.
7. Vytvoř nový adresový prostor. Je-li oblast textu již aktivní bude sdílená, jinak se inicializuje ze souboru.
8. Zkopíruj argumenty a proměnné okolí zpátky do uživatelského zásobníku.

9. Inicializuj HW kontext. Počítadlo instrukcí je nastaveno na adresu vstupního bodu programu.

Ukončení procesu - `exit()`

1. Ignoruj všechny signály.
2. Zavři všechny otevřené soubory.
3. Uvolni okamžitý adresář
4. Uvolni, je-li změněný kořenový adresář.
5. Uvolni přidělené paměťové oblasti.
6. Zapiš statistiky o využívání prostředků a parametr **`stav`** do záznamu **`proc.`**
7. Změň stav na **`mátoha`**.
8. PID rodiče všech potomků nastav na 1 – proces **`init`**
9. Pošli signál skončení potomka SIGCHLD rodiči.
10. Je-li rodič spící, vzbuď ho.
11. Vykonej přepnutí kontextu pro nový naplánovaný proces **`swtch()`**.

po skončení volání **`exit()`** je proces ve stavu **`mátoha`** a obsazuje záznam **`proc`**

Očekávaní skončení potomka - `wait()`

1. Chyba, nemá-li proces potomky.

2. Má-li potomka mátohu, vyber jednoho z nich, připočítej využití prostředků rodiči, uvolni záznam **`proc`** vrať PID potomka a stav skončení.
3. Jinak přejdi do stavu spící do příchodu signálu SIGCHLD.