Struktura jądra UNIX

Wykład 5 i 6: Organizacja jądra FreeBSD

Asynchroniczne zdarzenia obsługiwane przez jądro

Jądro reaguje na zdarzenia zewnętrzne pochodzące
od czasomierzy i innych urządzeń wejścia-wyjścia.

W wyniku obsługi przerwania jądro może:

1. odblokować zadanie
2. zmienić bieżąco wykonywane zadanie (**wywłaszczanie**)
3. wykonać procedurę obsługi
   a. przerwania urządzenia wejścia-wyjścia
   b. przekroczenia terminu (np. przekroczony czas oczekiwania na pakiet)

Synchroniczne zdarzenia obsługiwane przez jądro

Jądro pełni rolę interpretera instrukcji, których procesor nie mógł wykonać w danym kontekście. Jądro może zasymulować wykonanie instrukcji (**`trap`**), albo naprawić kontekst, w którym procesor chciał ją wykonać (**`fault`**).

Przez kontekst najczęściej mamy na myśli *user-space*. Jednakże błędy mogą się pojawiać również w *kernel-space*, np. pamięć jądra może być stronicowalna, albo błąd programisty jądra zakończy się *kernel panic*.

Z pułapką / błędem skojarzony jest kontekst procesora oraz dodatkowe informacje → jakie dla odwołania do pamięci?

Zadanie może oczekiwać na kilka sposobów

1. **Brak snu:** zadanie czeka na zwolnienie blokady wirującej.
2. **Sen ograniczony** (ang. *bounded sleep*):
   zadanie oczekuje na zwolnienie blokady.
   → jedyny zasób którego zadanie nie ma, to czas procesora
3. **Sen nieograniczony** (ang. *unbounded sleep*): zadanie oczekuje na zdarzenie zewnętrzne, które wydarzy się w niedalekiej przyszłości (intencja programisty jądra).
4. **Sen nieograniczony przerywalny** (ang. *unbounded interruptible sleep*): jak wyżej, ale istnieje możliwość wybudzania sygnałem uniksowym, np. `SIGINT`.

Szczegóły w `locking(9)`, ale przyjrzymy się temu później...

Jądro zarządza kontekstem wykonania

1. **Pełen kontekst procesora** zawiera rejestry ogólnego przeznaczenia, rejestry stanu (tryb pracy procesora, bieżąca maska przerwań) i ew. informacje o błędzie.
   → program wstrzymany wykonaniem przerwania lub pułapki
2. **Skrócony kontekst procesora** zawiera rejestry ogólnego przeznaczenia *caller-saved* (czemu?) i rejestry stanu.
   → kooperacyjna zmiana kontekstu w wyniku uśpienia na **kmutex**
3. Kontekst translacji adresów, czyli globalna **tablica stron jądra** i **tablica stron użytkownika** bieżącego procesu.
4. Stan przerwań, tj. *interrupt priority level* oraz maski **zablokowanych** i **oczekujących przerwań** urządzeń I/O.
   → rejestry masek mogą być odwzorowane w pamięć fizyczną

Obsługa zdarzeń asynchronicznych we FreeBSD

Zegar systemowy

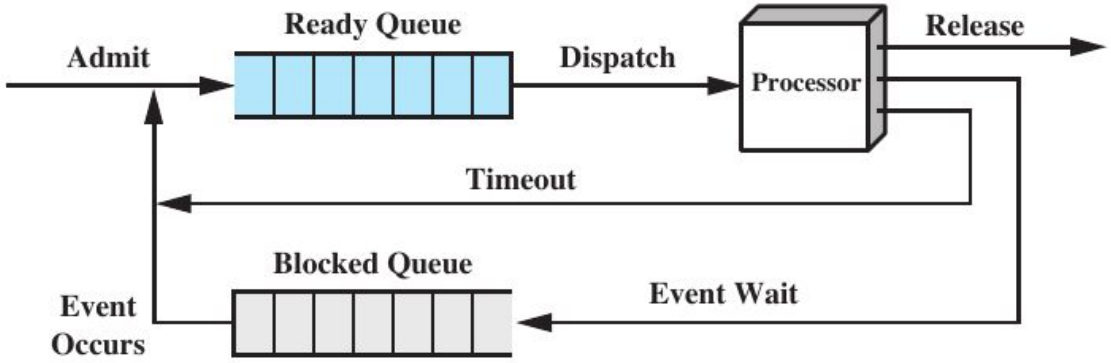
Odmierza **takty zegara** (FreeBSD: 1000Hz) od startu systemu.

- odmierza czas rzeczywisty (`hardclock`)
- wyzwala wywłaszczenie zadania (`sched_tick`)
- mierzenie czasu zużywanego przez proces (`sched_pstats`)
- wyzwala zdarzenia po upłynięciu terminu (`callout`)
- gromadzi statystyki dot. zużycia zasobów (`statclock`)
- służy do profilowania programów (`profclock`)
- zgłasza aktywność **licznikowi dozorującemu** (ang. *watchdog timer*)

Data, godzina, … w **zegarze czasu rzeczywistego** podtrzymywanego bateryjnie i pobierane w trakcie startu systemu → `inittodr`.

Z reguły `hardclock`, `profclock` i `statclock` są uruchamianie z inną częstotliwością. Co zrobić jeśli mamy tylko jeden czasomierz sprzętowy?

Wywłaszczanie



Wątki w stanie `READY` na **kolejce wątków uruchamialnych** (ang. *run queue*) → `FIFO` (ang. *first-in, first-out*).

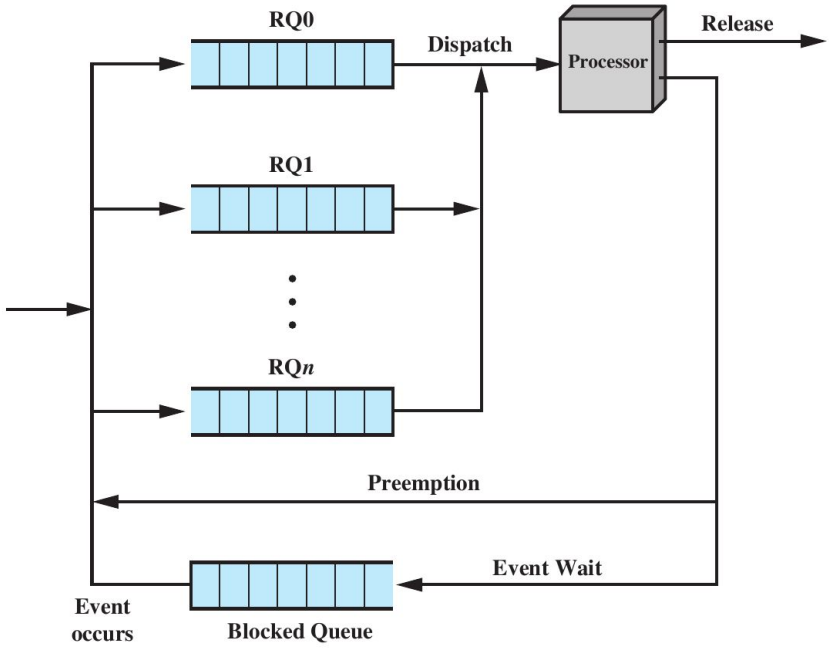
Wątek uruchomiony w globalnej zmiennej `curthread`. Obsługa przerwania zegarowego `statclock` woła `sched_clock`, które sprawdza czy kwant czasu się wyczerpał `SLICEEND`. Tak? Wątek oznaczamy flagą `NEEDRESCHED` i wracając z przerwania* `ast` przełączamy kontekst `mi_switch(9)` podając przyczynę `PREEMPT`.

Kolejki wątków uruchamialnych

Planista krótkoterminowy wybiera wątek o najwyższym **priorytecie**. Robi to >10k/s, więc musi być szybkie! Bitmapa oznaczająca niepustość kolejek i `ffs`.

Procedury planisty korzystające z `runq`, mogą być wołane w górnej i dolnej połówce. Zatem muszą być synchroniowane przez wyłączanie przerwań (w SMP *spinlock*'iem) !

Range	Class	Thread type
0 – 47	ITHD	bottom-half kernel (interrupt)
48 – 79	REALTIME	real-time user
80 – 119	KERN	top-half kernel
120 – 223	TIMESHARE	time-sharing user
224 – 255	IDLE	idle user

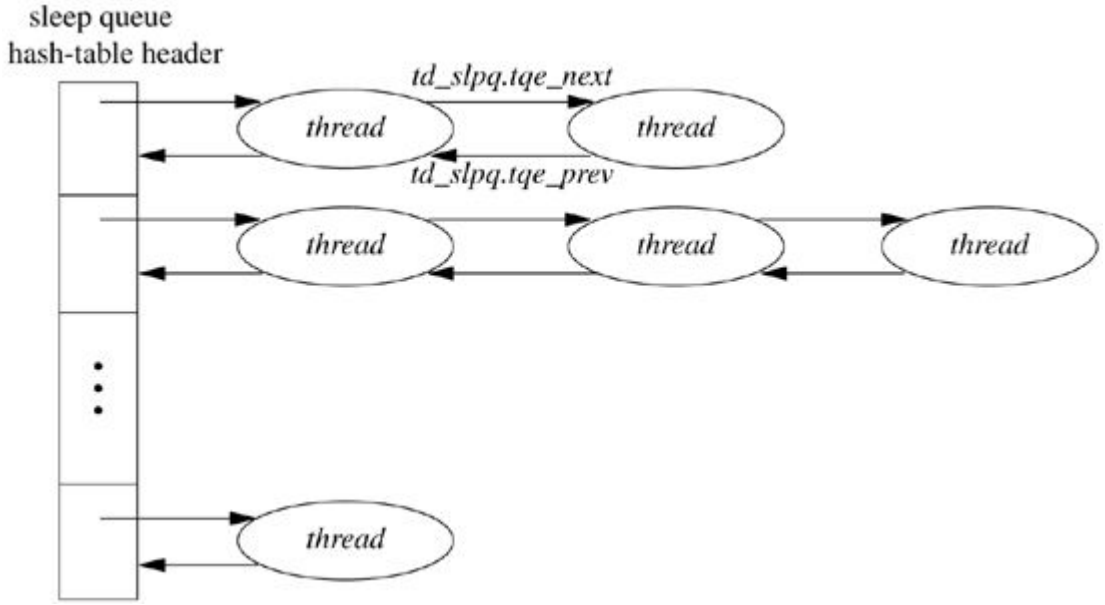


Kolejki uśpionych wątków

Wątki są usypiane na **kanałach oczekiwania** (`td_wchan`), których identyfikatorami są dowolne adresy. Procedura `sleepq_signal` wybudza wątek o najwyższym priorytecie lub najdłużej oczekujący.

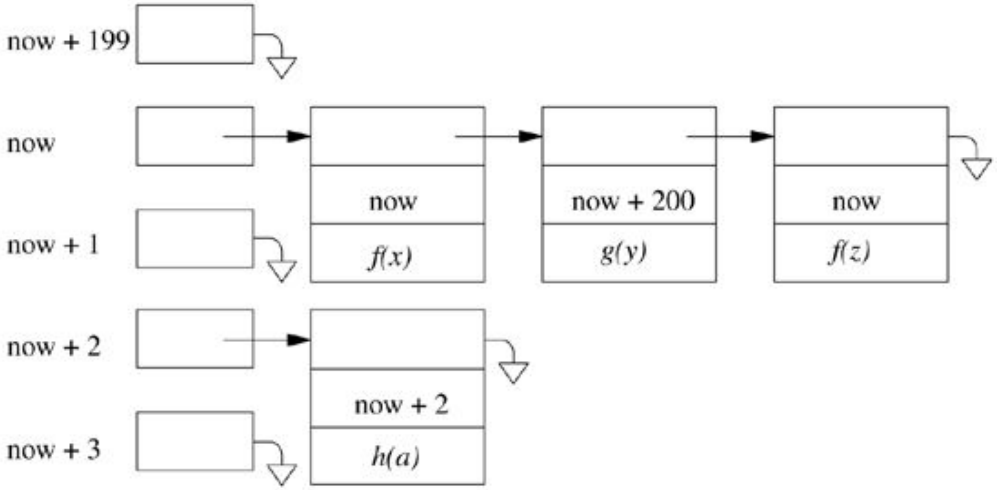
UWAGA! `wchan` to tylko etykieta, głowa listy wątków oczekujących na dane zdarzenie jest “przechowywana” gdzieś indziej!

Liczby zdarzeń, na które można oczekiwać jest dużo więcej niż wątków! … struktur `sleepqueue` jest tyle samo co wątków. Dlaczego?



Obsługa terminów

- symulacja wielu czasomierzy
- wybudzanie procesów (`alarm`)
- retransmisja zagubionych pakietów sieciowych
- liczniki dozorujące działanie systemu
- sterowniki urządzeń



Każdy z `N` kubełków przechowuje nieuporządkowaną listę procedur, które należy zawołać w czasie `t % N`. Za każdym razem przenosimy znacznik `now` tak by odzwierciedlał bieżący czas przy okazji obsługując zaległości. Interfejs `callout` w jądrach BSD implementuje **kolejki kalendarzowe**.

Hashed and Hierarchical Timing Wheels: Efficient Data Structures for Implementing a Timer Facility

FreeBSD: Główna procedura obsługi przerwań (1)

Po otrzymaniu przerwania procesor z wyłączonymi przerwaniami wchodzi do `MipsUserIntr` albo `MipsKernIntr`, jeśli odpowiednio został przerwany program wykonujący się w user / kernel-space.

W obydwu przypadkach będziemy zapisywać pełen kontekst procesora `trapframe`, więc dlaczego to rozróżnienie? Kontekst:

- **`user-space`** odkładamy w dobrze znane miejsce w **`td_pcb`**
  → przed powrotem do user-space czasami będziemy go modyfikować
- **`kernel-space`** po prostu odkładamy na stos jądra

UWAGA! Jądro nie korzysta z FPU (ang. *floating point unit*), zatem jego kontekstem musi zarządzać tylko przy przechodzeniu do user-space.

FreeBSD: Główna procedura obsługi przerwań (2)

W trakcie powrotu z przerwania do:

- **`user-space`** wykonujemy `ast` (ang. *asynchronous system trap*) (NetBSD: `userret(9)`) jeśli wątek miał ustawioną jedną z flag `TDF_NEEDRESCHED` lub `TDF_ASTPENDING` (tj. wywłaszczenie lub zdarzenie asynchroniczne)
- **`kernel-space`** wykonujemy procedurę `critical_exit` (NetBSD: `kpreempt_enable(9)`) zmieniającą kontekst, jeśli `td_owepreempt` było niezerowe (*preempt on last critical_exit*)

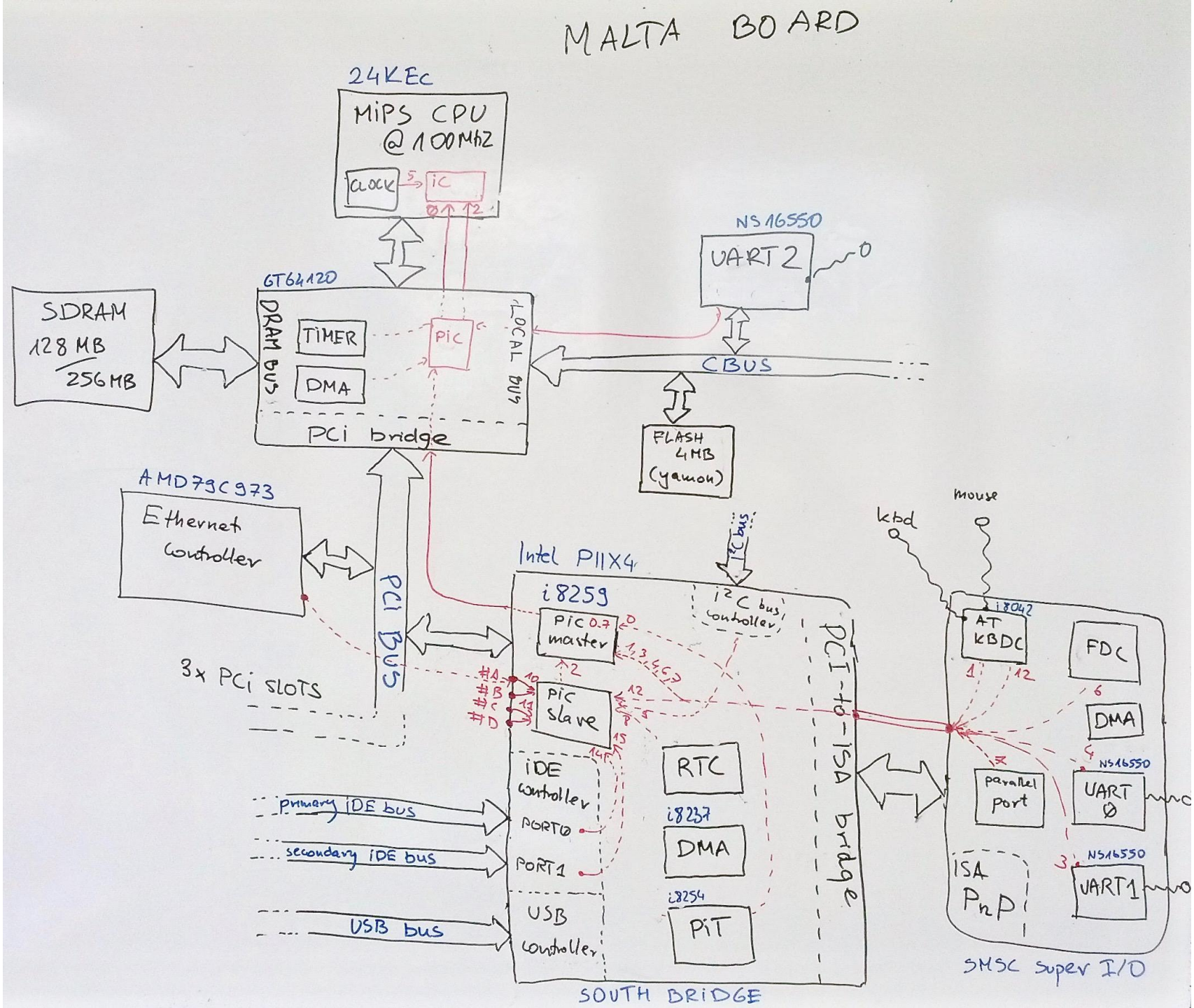
Dzięki `td_owepreempt` procedura może opóźnić żądanie zmiany kontekstu do ostatniego z zagnieżdżonych przerwań lub wyjścia z sekcji krytycznej w górnej połówce.

FreeBSD: Główna procedura obsługi przerwań (3)

Kod asemblerowy woła procedurę obsługi przerwań w języku C: zależną od architektury `cpu_intr` (`MIPS`) albo niezależną `intr_irq_handler` (`AArch64`).

Zbierają statystyki przerwań i przechodzą do wykonania `intr_event_handle` dla zdarzeń sprzętowych podpiętych do głównego kontrolera przerwań (zintegrowanego z procesorem).

Przerwania są zorganizowane w drzewiastą hierarchię.
W węzłach wewnętrznych mamy kontrolery przerwań,
a w liściach poszczególne urządzenia.


MALTA BOARD
24KEc
MIPS CPU
@100MHz
clock
iC
GT64120
SDRAM
128 MB
256MB
DRAM bus
TIMER
DMA
PIC
LOCAL bus
PCI bridge
NS16550
UART 2
CBUS
FLASH
4MB
(yamon)
AMD79C973
Ethernet
Controller
PCI Bus
3x PCI SLOTS
Intel PIIX4
i8259
PIC master
PIC slave
I2C bus
I2C bus controller
IDE
controller
PORT0
PORT1
primary IDE bus
secondary IDE bus
USB
controller
USB bus
RTC
i8237
DMA
i8254
PIT
SOUTH BRIDGE
PCI-to-ISA bridge
mouse
kbd
i8042
AT
KBDC
FDC
DMA
parallel
port
NS16550
UART 0
NS16550
UART 1
ISA
PnP
SMSC super I/O

FreeBSD: obsługa zdarzeń sprzętowych

1. Wybór łańcucha `intr_event` na podstawie numeru przerwania.
   → przechowuje listę rekordów opisujących sposób **obsługi zdarzeń**
2. Aktualizacja statystyk i wykrywanie **nawałnicy** (ang. *interrupt storm*).
3. Odpytywanie procedur `filter` poszczególnych `intr_handler` w poszukiwaniu urządzenia wymagającego obsługi zdarzenia.
4. `filter` obsłużyło całkowicie zdarzenie (`HANDLED`)
   → `ie_post_filter` sygnalizuje **EOI** (ang. end of interrupt)
5. `filter` oddelegowało (`SCHEDULE_THREAD`) część pracy `handler` do wykonania w kontekście **wątku przerwania** `ithread` (`pri` 0…47)
   → przy powrocie z obsługi przerwania przełączamy kontekst na `ithread`

WAŻNE! `filter` wykonuje się w dolnej połówce, a `handler` w górnej!

Czemu `filter` nie może używać blokad usypiających, a wirujące są ok?

FreeBSD: wątki obsługi przerwań

Wykonywanie całej obsługi przerwania w kontekście przerwanego wątku (tj. na jego stosie) jest niezręczne. Nie można korzystać z procedur, które mogą pójść spać → inaczej zablokujemy przerwany wątek… z wyłączonym przerwaniem!

Pójść spać mogą procedury, które przełączają kontekst, np. zakładanie blokad, czekanie na zmiennych warunkowych lub oczekiwanie na zdarzenia `tsleep`. Tj. nie można się odwoływać do podsystemów, które korzystają z blokad… czyli większości :-(

Proste zadania można wykonać w `filter`, ale bardziej złożone należy oddelegować do handler wykonującego się w kontekście wątku przerwania `ithread(9)`.

FreeBSD: Wątki przerwań

`ithread(9)` to wątki jądra wybudzane przez przyjście przerwań. Wątki przerwań mają priorytety wyższe niż wszystkie inne wątki.

Wykonują procedury obsługi przerwania używające blokad, ale dozwolone są tylko wirujące lub wchodzące w sen ograniczony → np. alokacja pamięci musi używać flagi **`M_NOWAIT`**. Dlaczego?

Dany wątek musi być skojarzony z dokładnie jednym `intr_event`.

Przed oddelegowaniem pracy do wątku przerwania wołana jest procedura `pre_ithread` wyłączająca przerwanie.

Po wybudzeniu wątek uruchamia procedury z listy posortowanej względem priorytetów. Po wykonaniu procedur obsługi wołana jest `post_ithread`, które aktywuje przerwanie.

Usypianie wątków we FreeBSD

Przełączanie zadań `mi_switch(9)`

`mi_switch` (ang. *machine independent*) wraz ze `sched_switch` aktualizują statystyki – liczbę zmian kontekstu, czas wątku spędzony na procesorze, itd. Następnie wybiera wątek docelowy `sched_choose`, na który przełączy kontekst `cpu_switch` (`MIPS`) nie dopuszczając do wywłaszczenia (wyłączone przerwania).

Przy przełączaniu kontekstu jądro wymienia przestrzeń adresową użytkownika przy pomocy `pmap_activate`. Czemu teraz?

Z `mi_switch` korzystają środki synchronizacji jądra (`sleepq_wait`, `turnstile_wait`), oddawanie sterowania `kern_yield`, wstrzymywanie sygnałem `SIGSTOP`, kończenie wątku oraz po obsłużeniu wyjątku procesora o ile zlecono wywłaszczenie.

Usypianie zadań

Usypianie zawsze zachodzi w sposób kooperacyjny, tj. wątek musi wywołać określoną procedurę. Żeby wejść w **sen:**

1. **ograniczony** na danej blokadzie, należy wykorzystać **bramki obrotowych** (ang. *turnstile*), a konkretniej `turnstile_wait`.
2. **nieograniczony** na punkcie oczekiwania, należy skorzystać ze `sleepqueue`, a dokładniej procedury `sleepq_wait`, która woła `mi_switch` z flagą `VOL`.
   → np. `wait` na zmiennej warunkowej
3. **nieograniczony przerywalny**, j.w. ale `sleepq_wait_sig`
   → procedury `*sleep(9)` przyjmujące argument `priority` `PCATCH`.

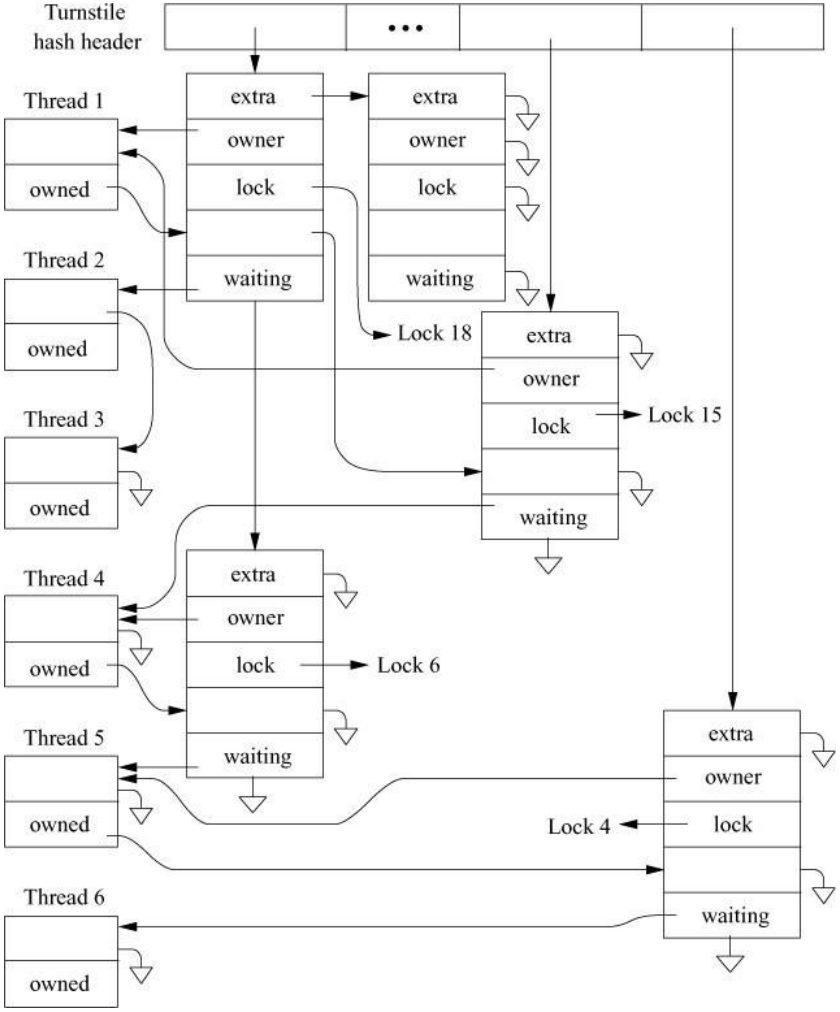
`mi_switch` odkłada na stos wątku jądra skrócony kontekst!

Bramki obrotowe (`turnstile`)

Jak zapobiec **inwersji priorytetów**? Używamy techniki **dziedziczenia priorytetów**! Będziemy śledzić właściciela blokady!

Przy wchodzeniu do sekcji krytycznej wątek o wyższym priorytecie zatrzymał się na blokadzie `turnstile_wait` → właścicielowi tymczasowo podbijamy priorytet `propagate_priority`.

Przy opuszczaniu sekcji krytycznej `turnstile_unpend` → przywracamy oryginalny priorytet.



Obsługa pułapek we FreeBSD

FreeBSD: Główna procedura obsługi wyjątków (1)

Po wygenerowaniu wyjątku procesor wchodzi do `MipsUserGenExc` lub `MipsKernGenExc` → działanie analogiczne do `MipsUserIntr` i `MipsKernIntr`, ale wołają procedurę `trap` (*machine dependent*).

Jeśli w przerwanym kontekście przerwania były:

- **włączone**: normalny tryb pracy w `user` lub `kernel-space`
  → wywołanie systemowe albo dostęp do pamięci stronicowalnej jądra
- **wyłączone**: przerwano sekcję krytyczną w górnej połówce albo wykonanie kodu w dolnej połówce (pod przerwaniem)
  → z reguły awaria jądra w kontekście, którego nie da się naprawić :-(

Większość pamięci jądra jest **zadrutowana** (ang. *wired memory*), a nie **stronicowalna** (ang. *pageable memory*).

FreeBSD: Główna procedura obsługi wyjątków (2)

Procedura `trap` analizuje przyczynę błędu zapisaną w `trapframe`:

1. Błędne odwołanie do pamięci:
   - emulacja bitów referenced-modified → `pmap_emulate_modified`
   - przekierowanie do podsystemu pamięci wirtualnej → `vm_fault_trap`
   - obsługa nieudanego `copyin` albo `copyout` → `pcb_onfault`
2. Emulacja niedostępnych instrukcji:
   - dostęp do pamięci pod niewyrównanym adresem
3. Wykonanie obsługi wywołania systemowego → `syscallenter`
4. Pozostałe wyjątki są zamieniane na sygnały → `trapsignal`

FreeBSD: Wywołania systemowe (1)

Zanim wykonamy procedurę wywołania systemowego trzeba wczytać jego numer oraz argumenty z rejestrów zapisanych w `trapframe` lub stosu (zależne od `ABI` i architektury).

Po wykonaniu procedury należy:

- ustawić w **`trapframe`** jej wynik i ew. **`errno`** jeśli wystąpił błąd
- zaktualizować wskaźnik instrukcji w **`trapframe`** na następną po instrukcji **`syscall`**, albo nie jeśli **`errno`** równe `ERESTART`
  → automatyczny restart przerwanych wywołań systemowych

Odpowiadają za to odpowiednio `cpu_fetch_syscall_args` oraz `cpu_set_syscall_retval` (`MIPS`).

FreeBSD: Wywołania systemowe (2)

Skrypt `makesyscalls.sh` na podstawie `syscalls.master` tworzy kilka plików źródłowych:

1. `syscall.h` definicje numerów wywołań systemowych
2. `sysproto.h` definicje struktur przechowujących argumenty
3. `init_sysent.c` tablica opisu wywołań systemowych `sysent`

```
3   AUE_READ    STD {
        ssize_t read(
            int fd,
            _Out_writes_bytes_(nbyte) void *buf,
            size_t nbyte
        );
    }
```

FreeBSD: Przykład wywołania systemowego

```
#define SYS_read    3
```

syscall.h

```
struct read_args {
    ...; int fd; ...;
    ...; void * buf; ...;
    ...; size_t nbyte; ...;
}
```

syscalls.master

```
{ AS(read_args),
  (sy_call_t *)sys_read,
  AUE_READ,
  NULL, 0, 0, ... },  /* 3 = read */
```

init_sysent.c

```
int sys_read(struct thread *td,
             struct read_args *uap)
{
  ...
}
```

sys_generic.c

FreeBSD: copyin i copyout

```
int sys_fstat(struct thread *td, struct fstat_args *uap) {
    struct stat ub;

    int error = kern_fstat(td, uap->fd, &ub);
    if (error == 0)
        error = copyout(&ub, uap->sb, sizeof(ub));
    return error;
}
```

Musi działać poprawnie dla pamięci stronicowalnej (błąd stron) i niedostępnej (zwraca EFAULT). No i musi być wydajne!

Ustawia **pcb_onfault** na kod, który zwraca EFAULT z procedury. Jego wartość zostanie adresem powrotu z obsługi wyjątku, jeśli **trap** nie uda się naprawić kontekstu wykonania instrukcji.

Pytania?