Salve prof! Abbiamo un bug veramente brutto e non sappiamo più dove mettere le mani. Chiediamo scusa per la domanda LUNGA, ci siamo assicurati di dividerla in sezioni così non perde la sanità mentale cercando punti diversi. Speriamo di non aver mancato un errore ovvio.

1. Struttura della domanda: Cercando di semplificare la sua lettura, scriviamo tendenzialmente nel modo in cui ci siamo approcciati al problema: prima la comprensione di cosa succede ad alto livello, poi le modifiche ai registri, seguite dai tentativi che abbiamo fatto per capire il motivo di tali modifiche e in fondo il codice (ridotto) su cui si basa la domanda. Alleghiamo inoltre:

    1. La domanda in un file testuale
    2. Il codice completo: non è esattamente "production-ready" ma speriamo risulti sufficientemente comprensibile. Se preferisce fare un ricevimento a noi va bene! NB: include alcune stampe di debug sul terminale 1 che abbiamo incluso noi, wrappate all'interno di un reset del timer per evitare di consumare tutto il `time slice`. Tutto ciò che menzioniamo in seguito è valido ANCHE senza queste stampe di debug. Più dettagli implementativi al codice di TRACE_SYSCALL.

2. Introduzione e nostra comprensione ad alto livello: Premessa doverosa: su uMPS il test procede e termina nel modo corretto (con la giusta sequenza di stampe e senza alcun errore, quantomeno apparente). Su uARM invece tutto procede senza intoppi fino allo startup del processo p5. Arrivati a questo punto il processo p4 ha già registrato ed eseguito con successo i custom handler della SPECPASSUP, ha risvegliato il processo `test` tramite la verhogen sul semaforo `endp4` e si appresta a stampare la stringa `p4 - try to redefine PGMVECT, it will cause p4 termination`. Dopo aver eseguito la WAITIO viene posto in attesa e l'esecuzione passa a p5 che cerca di acquisire la mutua esclusione sul terminale per stampare la stringa `p5 starts (and hopefully dies)` ma rimane correttamente bloccato in attesa di p4. p4 viene quindi risvegliato ma invece di continuare "normalmente" l'esecuzione della funzione `print`, sembra acquisire il contesto di p5 e stampare quindi il contenuto dell'argomento di `print` passato da p5. La stringa da stampare diventa quindi una risorsa condivisa e la stampa perde di senso: ma gli argomenti delle funzioni non dovrebbero sempre essere locali?

   P.S. la parte del test relativa alla GETCPUTIME non è ancora stata implementata ma non pensiamo possa essere in qualche modo legata al nostro problema.

3. Momento clou e comprensione a basso livello: All'entrata di p5 (attivando il breakpoint su p5), lo stato del processore sembra essere quello corretto di p5, ma eseguendo qualche step, prima della chiamata della system call passeren, nel processore vengono inseriti i valori del frame pointer e dello stack pointer di p4 (quando dovrebbero invece essere presenti quelli di p5).

4. Tentativi e modifiche che abbiamo tentato:

    1. Se nel test commentiamo la riga che esegue la createProcess di p5, il test viene completato con successo (ovviamente al netto dell'esecuzione di p5).
    2. Se utilizziamo lo stato di p4 per inizializzare p6, anche p6 causa un errore simile. Pur non andando in PANIC infatti l'esecuzione viene alterata al punto che il codice di p4 viene rieseguito una seconda volta.

    3. Se eseguiamo il test così com'è, abbiamo il seguente output:

```
 p4 starts p4 - try to cause a pgm trap access some non-existent memory
pgmTrapHandler - Access non-existent memory
p4a - try to generate a TLB exception
memory management (tlb) trap
p4 - try call sys13 to verify pass up
p4b - Invoking custom system call 13
Custom system call handler called successfully; continuing...
p1 knows p4 ended
p5 starts (and hopefully dies)
Custom system call handler called successfully; continuing...
p6 starts (and hopefully dies)
error: p5 alive after SYS13() with no trap vector
KERNEL PANIC!
```

    In questo caso, p5 sembra partire correttamente, e l'errore relativo alla SYS13 sembrerebbe segnalare l'errore di non aver terminato il processo in seguito ad una syscall non gestita da un passup. 4d. Tuttavia, se modifichiamo le stringhe stampate da p4 e p5 come segue, abbiamo un altro comportamento.

```
 //p4 prima
SYSCALL(VERHOGEN, (int)&endp4, 0, 0); /* V(endp4) */
print_test("p4 - try to redefine PGMVECT, it will cause p4 termination\n");
// p4 dopo
SYSCALL(VERHOGEN, (int)&endp4, 0, 0); /* V(endp4) */
print_test("4p4 - try to redefine PGMVECT, it will cause p4 termination\n");
//p5 prima
void p5() {
    print_test("p5 starts (and hopefully dies)\n");
//p5 dopo
void p5() {
    print_test("5p5 starts (and hopefully dies)\n");
```

    L'output in questo caso è il seguente:

```
 p4b - Invoking custom system call 13
Custom system call handler called successfully; continuing...
p1 knows p4 ended
4KERNEL PANIC!
```

    Sembra dunque che nel primo caso la prima 'p' stampata sia quella di p4, poi avvenga uno switch di frame e stack pointer su p5, e la stampa continua sul secondo carattere di p5. Nel secondo caso, invece, avendo stampato un 4, il controllo sul carattere stampato ritorna un errore poichè secondo p5, il primo carattere dovrebbe invece corrispondere ad un 5.

4. Se togliamo il controllo sul carattere stampato (che manda in panic il kernel) otteniamo l'output seguente:

```
 p4 starts
p4 - try to cause a pgm trap access some non-existent memory
pgmTrapHandler - Access non-existent memory
p4a - try to generate a TLB exception
memory management (tlb) trap
p4 - try call sys13 to verify pass up
p4b - Invoking custom system call 13 Custom system call handler called successfully; continuing...
p1 knows p4 ended
4p5 starts (and hopefully dies)
Custom system call handler called successfully; continuing...
p6 starts (and hopefully dies)
error: p5 alive after SYS13() with no trap vector KERNEL PANIC!
```

   Qui si nota come '4p5' metta in evidenza il fatto che p4 prenda ad un certo punto il frame pointer di p5. Infatti, alla SYS13 non termina il processo, perchè in realtà quello in esecuzione è p4, che aveva già registrato i passup. Esaminando inoltre la ready queue in ogni istante (impiegando svariate ore di debugging) sappiamo per certo che, mentre viene eseguito il codice di p5 da parte di p4, p5 non è nella ready queue, ma è bloccato sul mutex del terminale, e p4 è in testa alla ready queue, evidenziando il fatto che è p4 ad essere in esecuzione.

5. Relevant code: Per (forse) comodità, includiamo una versione ridotta del codice. Il codice intero è allegato. Le stampe di debug sono implementate con questa macro, che resetta e quindi "ackowledgwea" l'interrupt solo se l'interrupt è avvenuto durante la stampa:

```
#define TRACE_SYSCALL(fmt, ...)                                 \
{                                                               \
    bool toReset = !(INT_IS_PENDING(getCAUSE(), IL_TIMER)); \
    uint32_t time = getTIMER();                                 \
    debugln(fmt, ##__VA_ARGS__);                                \
    printReadyQueue();                                          \
    if (toReset)                                                \
        setTIMER(time);                                         \
}
```

   All'entrata di ogni handler, il processo corrente e la old area vengono salvate così:

```
#define ENTER_HANDLER(oldarea)         \
    pcb_t *current = getCurrent();     \
    state_t *old = (state_t *)oldarea; \
    current->p_s = *old;               \
    old = &current->p_s
```

   Il registro cause è ottenuto tramite:

```
#define CAUSE_GET(state) ((state)->CP15_Cause)
```

   L'handler delle syscall è strutturato quindi come uno switch sul numero della syscall:

```
void syscallHandler(void) {
    ENTER_HANDLER(SYSBK_OLDAREA);
    uint32_t cause = CAUSE_GET(old);
    if (CAUSE_IS_SYSCALL(cause)) {
#ifdef TARGET_UMPS
        PC_SET(old, PC_GET(old) + WORD_SIZE);
#endif
        uint32_t no = REG_GET(old, A0);

        switch (no) {
            case GETCPUTIME: {
                /* irrelevant code */
            }
            case CREATEPROCESS: {
                state_t *state = REG_GET(old, A1);
                int priority = REG_GET(old, A2);
                void **cpid = REG_GET(old, A3);
                SYSCALL_RETURN(old, createProcess(state, priority, cpid));
                printReadyQueue();
                if (current == getCurrent())
                    LDST(old);
                else
                    start();
```

```
                break;
            }
            case TERMINATEPROCESS: {
                pcb_t *pid = REG_GET(old, A1);
                SYSCALL_RETURN(old, terminateProcess(pid));
                printReadyQueue();
                if (pid)
                    LDST(old);  // No need to go through scheduler since process is using its time slice fairly
                else
                    start();
                break;
            }
            case VERHOGEN: {
                /* irrelevant code */
            }
            case PASSEREN: {
                /* irrelevant code */
            }
            case WAITIO: {
                /* irrelevant code */
            }
            case SPECPASSUP: {
                spu_t type = REG_GET(old, A1);
                state_t *spuOld = REG_GET(old, A2);
                state_t *spuNew = REG_GET(old, A3);
                SYSCALL_RETURN(old, specPassUp(type, spuOld, spuNew));
                LDST(old);
                break;
            }
            case GETPID: {
                /* irrelevant code */
            }
            default:
                if (current->sysbk_new == NULL) {
                    terminateProcess(NULL);
                    printReadyQueue();
                    start();
                } else {
                    *(current->sysbk_old) = *old;
                    printReadyQueue();
                    LDST(current->sysbk_new);
                }
        }
    }
}
```

In particolare, riportiamo qui il codice di createProcess(...), terminateProcess(...), e specPassUp(...):

```
syscall_ret_t createProcess(state_t *state, int priority, void **cpid) {
    pcb_t *p = allocPcb();
    if (p == NULL)
        return SYSCALL_FAILURE;
    p->p_s = *state;
    p->original_priority = priority;
    p->priority = priority;
    debugln("Created process %p (pc = %p)", p, p->p_s.pc);
    pcb_t *current = getCurrent();
    insertChild(current, p);
    addToReadyQueue(p);
    if (cpid)
        *cpid = p;
    return SYSCALL_SUCCESS;
}

syscall_ret_t terminateProcess(pcb_t *pid) {
    if (pid == NULL)
        pid = getCurrent();
    if (pid == NULL)
        return SYSCALL_FAILURE;
    println("about to kill %p", pid);
    killProgeny(pid);
    return SYSCALL_SUCCESS;
}
```

Dove killProgeny(...) è

```
void killProgeny(pcb_t *pid) {
    pcb_t *it;
    list_for_each_entry(it, &pid->p_child, p_sib) {
        killProgeny(it);
    }
    outChild(pid);
    // remove from semaphores (no need for V()
    // because value changes after process resumes)
    outBlocked(pid);
    // remove from readyQueue
    if (outProcQ(&readyQueue, pid))
        freePcb(pid);
}
```

E in ultimo

```
 /* This macro should only be used inside a function that returns syscall_ret_t */
#define REGISTER_SPU_HANDLER(p, field, old, new) \
    {                                            \
        if (p->field##_new != NULL) {            \
            terminateProcess(p);                 \
            return SYSCALL_FAILURE;              \
        }                                        \
        p->field##_old = old;                    \
        p->field##_new = new;                    \
    }

syscall_ret_t specPassUp(spu_t type, state_t *old, state_t *new) {
    pcb_t *p = getCurrent();
    switch (type) {
        case SPU_SYSCALL_BRK:
            REGISTER_SPU_HANDLER(p, sysbk, old, new);
            break;
        case SPU_TLB:
            REGISTER_SPU_HANDLER(p, tlb, old, new);
            break;
        case SPU_TRAP:
            REGISTER_SPU_HANDLER(p, trap, old, new);
            break;
    }
    return SYSCALL_SUCCESS;
}
```

6. Ringraziamenti: Complimenti per essere arrivato fin qui! La ringraziamo dell'attenzione e speriamo ci possa guidare nella risoluzione di questo fastidiosissimo errore