Analisi Comparativa tra:

OS161 e Xv6

corso di “Programmazione di sistema”

a.a. 2023/2024

Politecnico di Torino

A cura di:

Scaturro Andrea Pio: s319480@studenti.polito.it

Interrante Bonadia Alekos: s319849@studenti.polito.it

Introduzione

Nella seguente relazione è stata effettuata un analisi delle principali caratteristiche del sistema operativo xv6 evidenziando le differenze con il sistema operativo os161, trattato nel corso :“Programmazione di sistema”.

Il sistema operativo Xv6 è modellato sulla versione 6 di Unix implementato su architetture RISC-V. Ha un'architettura monolitica, dove tutte le funzionalità del kernel sono implementate in uno spazio di indirizzi comune .Include un sistema di gestione dei processi che consente la creazione, l'esecuzione e la terminazione di processi. Questo è essenziale per il supporto di esecuzione concorrente di programmi. Implementa un sistema di gestione della memoria che supporta l'allocazione e la deallocazione della memoria. Utilizza una tabella delle pagine per gestire la mappatura tra gli indirizzi virtuali e fisici, adattata alle specifiche della piattaforma RISC-V.

Fornisce supporto per le operazioni di I/O tramite chiamate di sistema, consentendo ai processi di comunicare con il sistema operativo e con altri processi.

Xv6 viene eseguito su un microprocessore RISC-V multi-core e gran parte delle sue funzionalità di basso livello (ad esempio, l'implementazione del processo) sono specifiche di RISC-V. RISC-V è una CPU a 64 bit e xv6 è scritto in "LP64" C, con long (L) e puntatori (P),che nel linguaggio di programmazione C sono di 64 bit, mentre int è 32 bit.

La CPU è circondata da hardware di supporto, in gran parte sotto forma di interfacce I/O. Xv6 è scritto per l'hardware di supporto simulato di qemu. Ciò include RAM, una ROM contenente il codice di avvio, una connessione seriale alla tastiera/schermo dell'utente e un disco per l'archiviazione.

RISC-V ha tre modalità in cui la CPU può eseguire istruzioni: modalità macchina, modalità supervisore e modalità utente.

Le istruzioni eseguite in modalità macchina hanno pieni privilegi; una CPU si avvia in modalità macchina che è quindi destinata principalmente alla configurazione di un computer.

In modalità supervisore la CPU può eseguire istruzioni privilegiate ad esempio, abilitare e disabilitare gli interrupt.

Kernel

Come già detto xv6 si basa su kernel monolitico ,quindi l'intero sistema operativo risiede nel kernel, in modo che le implementazioni di tutte le chiamate di sistema vengano eseguite in modalità supervisore .

In questa organizzazione l'intero sistema operativo viene eseguito con privilegi hardware completi. Questa organizzazione è comoda perché il progettista del sistema operativo non deve decidere quale parte del sistema operativo non necessita dei privilegi hardware completi. Inoltre, è più facile cooperare tra le diverse parti del sistema operativo.

Uno svantaggio dell'organizzazione monolitica è che le interfacce tra le diverse parti del sistema operativo sono spesso complesse, e quindi è facile per uno sviluppatore di sistema operativo commettere un errore. In un kernel monolitico, un errore è fatale, perché un errore in modalità supervisore spesso causa il fallimento del kernel. Se il kernel fallisce, tutte le applicazioni falliscono ed il computer deve riavviarsi per ricominciare.

Per ridurre il rischio di errori nel kernel, i progettisti del sistema operativo possono ridurre al minimo la quantità di codice del sistema operativo eseguito in modalità supervisore ed eseguire la maggior parte del sistema operativo in modalità utente.Questa organizzazione del kernel è chiamata microkernel ed è stata implementata per il sistema operativo os161.

In un microkernel, l'interfaccia del kernel consiste di alcune funzioni di basso livello per l'avvio di applicazioni, l'invio di messaggi, l'accesso all'hardware del dispositivo, ecc. Questa organizzazione consente al kernel di essere relativamente semplice, poiché la maggior parte del sistema operativo risiede a livello utente.

Dato che xv6 non fornisce molti servizi, il suo kernel è più piccolo di altri microkernel, ma concettualmente xv6 è monolitico.

Gestione della memoria

Le istruzioni RISC-V (sia utente che kernel) manipolano gli indirizzi virtuali. La RAM è indicizzata con indirizzi fisici. L'hardware della tabella delle pagine RISC-V collega questi due tipi di indirizzi, mappando ciascun indirizzo virtuale su un indirizzo fisico.

Xv6 funziona su Sv39 RISC-V, il che significa che vengono utilizzati solo gli ultimi 39 bit di un indirizzo virtuale a 64 bit; i primi 25 bit non vengono utilizzati.

In questa configurazione Sv39, una tabella delle pagine RISC-V è logicamente un array di 2^{27} entry della tabella delle pagine (PTE). Ogni PTE contiene un numero di pagina fisica (PPN) a 44 bit e alcuni flag. L'hardware di paging traduce un indirizzo virtuale utilizzando i primi 27 bit dei 39 bit da indicizzare nella tabella delle pagine per trovare un PTE e creando un indirizzo fisico a 56 bit i cui primi 44 bit provengono dal PPN nel PTE e la cui parte inferiore di 12 bit vengono copiati dall'indirizzo virtuale originale. La Figura 1.1 mostra questo processo con una vista logica della tabella delle pagine come un semplice array di PTE .

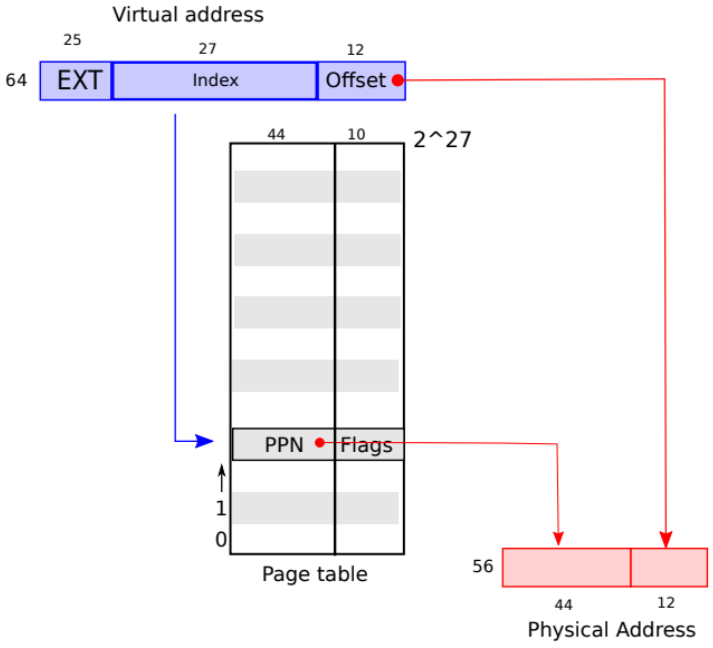


Figura 1.1

Una tabella delle pagine fornisce al sistema operativo il controllo sulle traduzioni degli indirizzi da virtuale a fisico con la granularità di blocchi allineati di 4096 (2^{12}) byte (pagina).

Anche l'indirizzo fisico ha spazio di crescita: nel formato PTE c'è spazio per far crescere il numero della pagina fisica di altri 10 bit.

Una tabella delle pagine viene archiviata nella memoria fisica come un albero a tre livelli. La radice dell'albero è una pagina della tabella delle pagine da 4096 byte che contiene 512 PTE, che contengono gli indirizzi fisici per le pagine della tabella delle pagine nel livello successivo dell'albero. Ciascuna di queste pagine contiene 512 PTE per il livello finale nell'albero. L'hardware di paginazione utilizza i primi 9 bit dei 27 bit per selezionare un PTE nella pagina radice della tabella delle pagine, i 9 bit centrali per selezionare un PTE in una pagina della tabella delle pagine nel livello successivo dell'albero e gli ultimi 9 bit bit per selezionare il PTE finale.

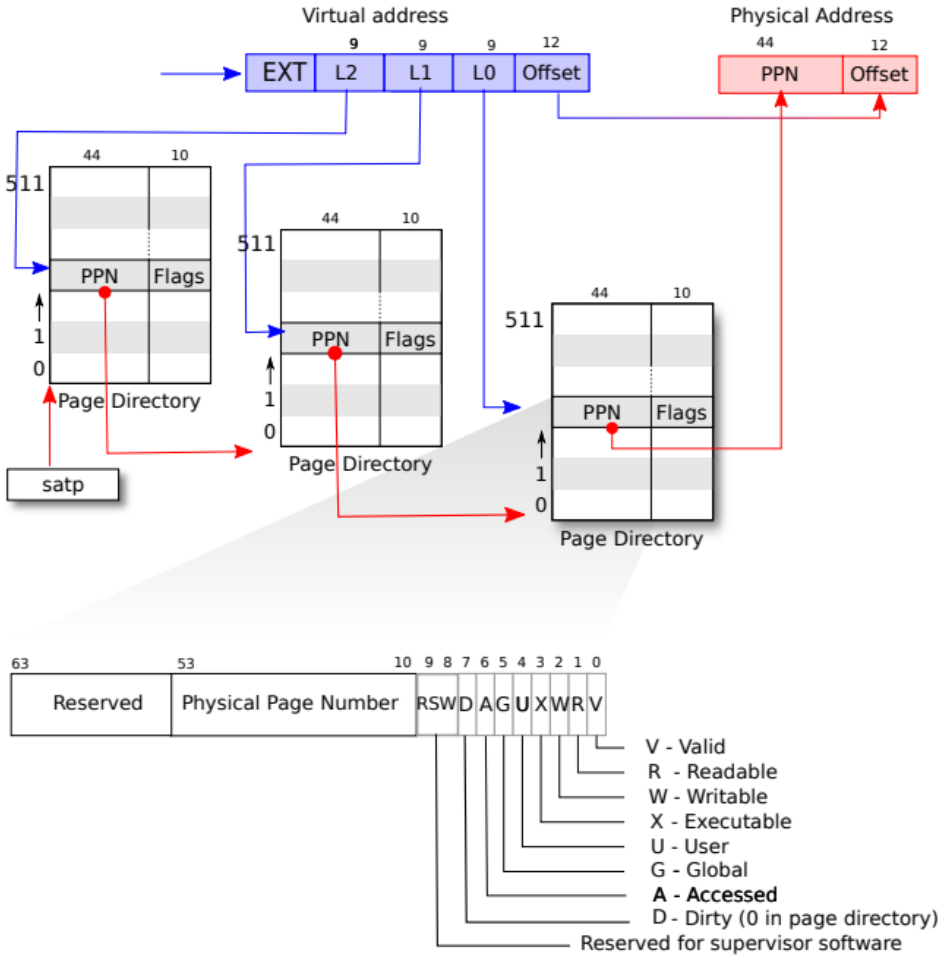
Se uno qualsiasi dei tre PTE richiesti per tradurre un indirizzo non è presente, l'hardware di paging solleva un'eccezione page-fault, lasciando al kernel il compito di gestire l'eccezione.

La struttura a tre livelli della Figura 1.2 consente un modo efficiente in termini di memoria per registrare i PTE, rispetto alla progettazione a livello singolo della Figura 3.1. Nel caso comune in cui ampi intervalli di indirizzi virtuali non hanno mappature, la struttura a tre livelli può omettere intere directory di pagine. Ad esempio, se un'applicazione utilizza solo poche pagine a partire dall'indirizzo zero, allora le voci da 1 a 511 della directory delle pagine di livello superiore non sono valide e il kernel non deve allocare le pagine per le directory delle pagine intermedie 511. Inoltre, il kernel non deve allocare pagine per le directory delle pagine di livello inferiore per quelle 511 directory delle pagine intermedie. Pertanto la progettazione a tre livelli salva 511 pagine per le directory di pagine intermedie e 511 × 512 pagine per le directory di pagine di livello inferiore.

Sebbene una CPU percorra la struttura a tre livelli nell'hardware come parte dell'esecuzione di un'istruzione di caricamento o di archiviazione, un potenziale svantaggio di tre livelli è che la CPU deve caricare tre PTE dalla memoria per eseguire la traduzione dell'indirizzo virtuale nell'istruzione di caricamento/archiviazione. istruzione ad un indirizzo fisico.

Per evitare il costo del caricamento di PTE dalla memoria fisica, una CPU RISC-V memorizza nella cache le voci della tabella delle pagine in un Translation Look-aside Buffer (TLB).

Figura 1.2: Dettagli di traduzione dell'indirizzo RISC-V.



Ciascun PTE contiene bit di flag che indicano all'hardware di paging come è consentito utilizzare l'indirizzo virtuale associato. PTE_V indica se la PTE è presente: se non è impostata, un riferimento alla pagina provoca un'eccezione (cioè non è consentito). PTE_R controlla se le istruzioni possono leggere la pagina.

PTE_W controlla se le istruzioni possono scrivere nella pagina. PTE_X controlla se la CPU può interpretare il contenuto della pagina come istruzioni ed eseguirle.

PTE_U controlla se le istruzioni in modalità utente possono accedere alla pagina; se PTE_U non è impostato, la PTE può essere utilizzata solo in modalità supervisore.

La Figura 1.2 mostra come funziona il tutto.

I flag e tutte le altre strutture relative all'hardware della pagina sono definiti in (kernel/riscv.h).

Per dire a una CPU di utilizzare una tabella delle pagine, il kernel deve scrivere l'indirizzo fisico della tabella delle pagine root nel registro satp. Una CPU tradurrà tutti gli indirizzi generati dalle istruzioni successive utilizzando la tabella delle pagine puntata dal proprio satp. Ogni CPU ha il proprio satp in modo che diverse CPU possano eseguire processi diversi, ciascuna con uno spazio di indirizzi privato descritto dalla propria tabella delle pagine.

Tipicamente un kernel mappa tutta la memoria fisica nella sua tabella delle pagine in modo che possa leggere e scrivere qualsiasi posizione nella memoria fisica utilizzando le istruzioni di caricamento/archiviazione. Poiché le directory delle pagine si trovano nella memoria fisica, il kernel può programmare il contenuto di una PTE in una directory della pagina scrivendo nell'indirizzo virtuale della PTE utilizzando un'istruzione di memorizzazione standard.

Vediamo che analogamente alla gestione delle pagine in os161, in xv6 è implementata una gestione gerarchica di page table ma in un contesto multicore.

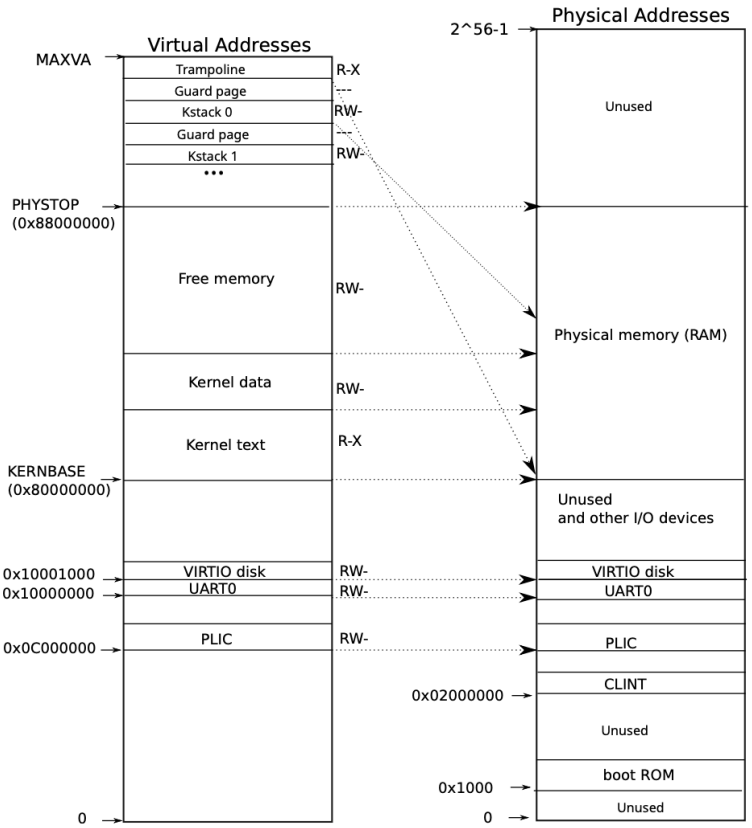


Figura 1.3: A sinistra, lo spazio degli indirizzi del kernel di xv6. RWX si riferisce a lettura, scrittura ed esecuzione PTE autorizzazioni. A destra, lo spazio degli indirizzi fisici RISC-V che xv6 si aspetta di vedere.

XV6, utilizza una struttura di gestione della memoria che comprende una tabella delle pagine per ciascun processo in esecuzione. Ogni processo ha la propria tabella delle pagine, la quale descrive come lo spazio degli indirizzi virtuale dell'utente è mappato su quello fisico della memoria.

Inoltre, c'è una singola tabella delle pagine che è dedicata allo spazio degli indirizzi del kernel. Questa tabella permette al kernel di mappare e accedere alle risorse fisiche della memoria e ai dispositivi hardware a indirizzi virtuali prevedibili, fornendo una struttura organizzata per la gestione delle risorse di sistema.

In un sistema con indirizzi virtuali prevedibili, il kernel del sistema operativo e i processi utente sanno in anticipo a quale indirizzo virtuale corrisponderà una certa porzione di memoria fisica.

Questo approccio consente a XV6 di garantire che ogni processo abbia un controllo isolato sulla propria memoria virtuale, fornendo al contempo al kernel un accesso strutturato e sicuro alle risorse di sistema. La tabella delle pagine è quindi un componente chiave nella gestione della memoria di XV6, contribuendo alla sicurezza e all'efficienza del sistema operativo.

QEMU simula un computer che include RAM (memoria fisica) a partire dall'indirizzo fisico 0x80000000 e continuando fino a 0x88000000 (PHYSTOP).
La simulazione QEMU include anche dispositivi I/O come un'interfaccia disco. QEMU espone le interfacce del dispositivo al software come registri di controllo mappati in memoria che si trovano sotto 0x80000000 nello spazio degli indirizzi fisici. Il kernel può interagire con i dispositivi leggendo/scrivendo questi indirizzi fisici speciali; tali letture e scritture comunicano con l'hardware del dispositivo anziché con la RAM.

Il kernel accede alla RAM e ai registri del dispositivo mappati in memoria utilizzando la "mappatura diretta"; ovvero mappare le risorse su indirizzi virtuali uguali all'indirizzo fisico. Ad esempio, il kernel stesso si trova in KERNBASE=0x80000000 sia nello spazio degli indirizzi virtuali che nella memoria fisica. La mappatura diretta semplifica il codice del kernel che legge o scrive la memoria fisica. Ad esempio, quando il fork alloca la memoria utente per il processo figlio, l'allocatore restituisce l'indirizzo fisico di quella memoria; fork utilizza quell'indirizzo direttamente come indirizzo virtuale quando copia la memoria utente del genitore sul figlio.

Esistono un paio di indirizzi virtuali del kernel che non sono mappati direttamente:

- trampoline page.

  Ogni processo in esecuzione ha la sua tabella delle pagine che gestisce la corrispondenza tra indirizzi virtuali e fisici della memoria.La trampoline page è mappata in tutte le tabelle delle pagine dei processi a un indirizzo noto, chiamato TRAMPOLINE. Questo indirizzo si trova nella parte superiore dello spazio degli indirizzi virtuali.

  Quando si verifica una trap o un'eccezione, il controllo passa al gestore delle trap, che si trova nel kernel. La pagina del trampolino gioca un ruolo chiave in questo processo. È mappata anche nella tabella delle pagine del kernel allo stesso indirizzo TRAMPOLINE.

  Ora, il fatto che la pagina del trampolino sia mappata nelle tabelle delle pagine utente senza il flag PTE_U significa che è accessibile solo in modalità

supervisore (kernel). Questo è utile perché il gestore delle trap, che può iniziare l'esecuzione dalla pagina del trampolino, deve essere eseguito con privilegi più elevati rispetto al normale codice utente.

Quando si verifica una trap, il controllo può iniziare nell'area del trampolino, che è accessibile in modalità supervisore. Da lì, il gestore delle trap può continuare l'esecuzione, e poiché la pagina del trampolino è anche mappata nella tabella delle pagine del kernel allo stesso indirizzo, il gestore può passare fluidamente al kernel senza dover effettuare cambiamenti significativi agli indirizzi.

In sostanza, la pagina del trampolino funge da ponte tra il livello utente e il livello kernel durante le trap, garantendo una gestione fluida delle transizioni tra i due livelli di privilegio del sistema operativo.

• Le pagine dello stack del kernel.

Ogni processo ha il proprio stack del kernel, che è mappato in alto in modo che sotto di esso xv6 possa lasciare una pagina di guardia non mappata. Il PTE della pagina di guardia non è valido (cioè, PTE_V non è impostato), quindi se il kernel va in overflow di uno stack del kernel, probabilmente causerà un'eccezione e il kernel andrà in panic. Senza una pagina di guardia uno stack in overflow sovrascriverebbe altra memoria del kernel, determinando un funzionamento errato.

Sebbene il kernel utilizzi i suoi stack tramite mappature con memoria elevata, essi sono accessibili al kernel anche tramite un indirizzo mappato direttamente.

Il kernel mappa le pagine destinate al "trampoline" e al codice del kernel con i permessi PTE_R (Read) e PTE_X (Execute). Il kernel legge ed esegue istruzioni da queste pagine. Le pagine destinate ad altre operazioni sono mappate con i permessi PTE_R e PTE_W (Write), così da poter leggere e scrivere nella memoria di queste pagine. Le mappature per le "guard pages" (pagine di protezione) sono invalide.

Quindi le pagine destinate al "trampoline" e al codice del kernel possono solo essere lette (PTE_R) ed eseguite (PTE_X), poiché contengono istruzioni eseguibili. Al contrario, le altre pagine possono essere lette e scritte (PTE_R e PTE_W) in quanto sono utilizzate per dati e variabili che il kernel può modificare.

Le "guard pages" hanno mappature invalide, il che significa che il loro accesso è proibito. Queste pagine possono essere utilizzate per creare uno spazio vuoto tra regioni di memoria per proteggere contro accessi imprevisti o errori di programmazione. In breve, questa configurazione è parte della gestione della

memoria del kernel, controllando attentamente come il kernel può accedere e utilizzare le diverse pagine di memoria.

Xv6 utilizza la memoria fisica tra la fine del kernel e PHYSTOP per l'allocazione del runtime. Assegna e libera intere pagine da 4096 byte alla volta. Tiene traccia di quali pagine sono libere inserendo una linked list attraverso le pagine stesse. L'assegnazione consiste nel rimuovere una pagina dalla linked list; la liberazione consiste nell'aggiungere alla lista la pagina liberata.

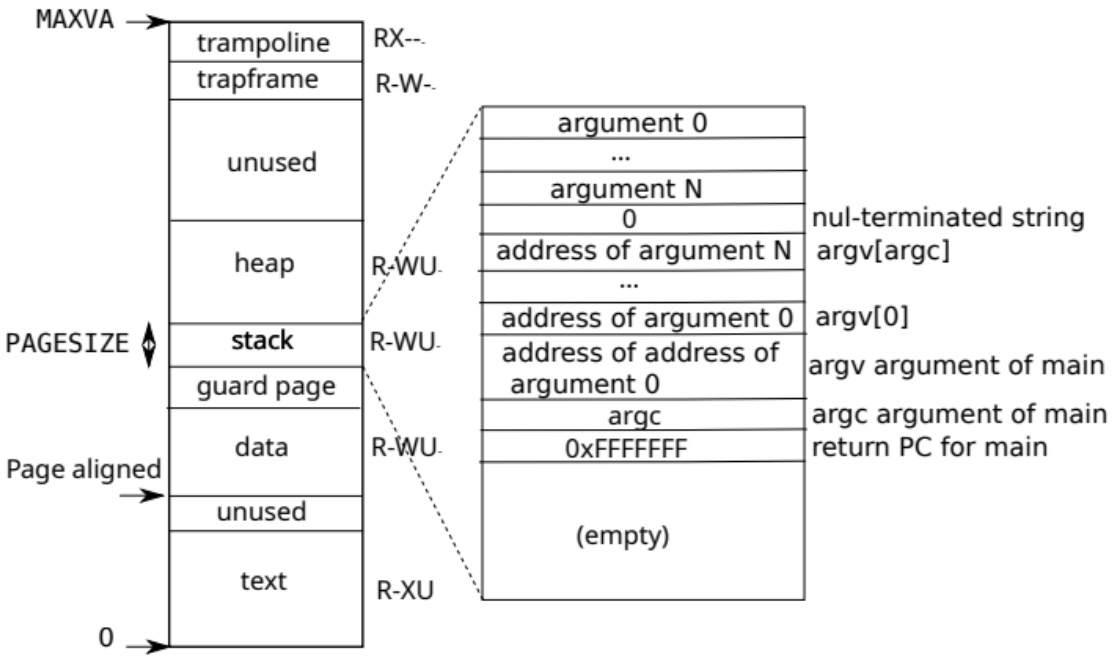
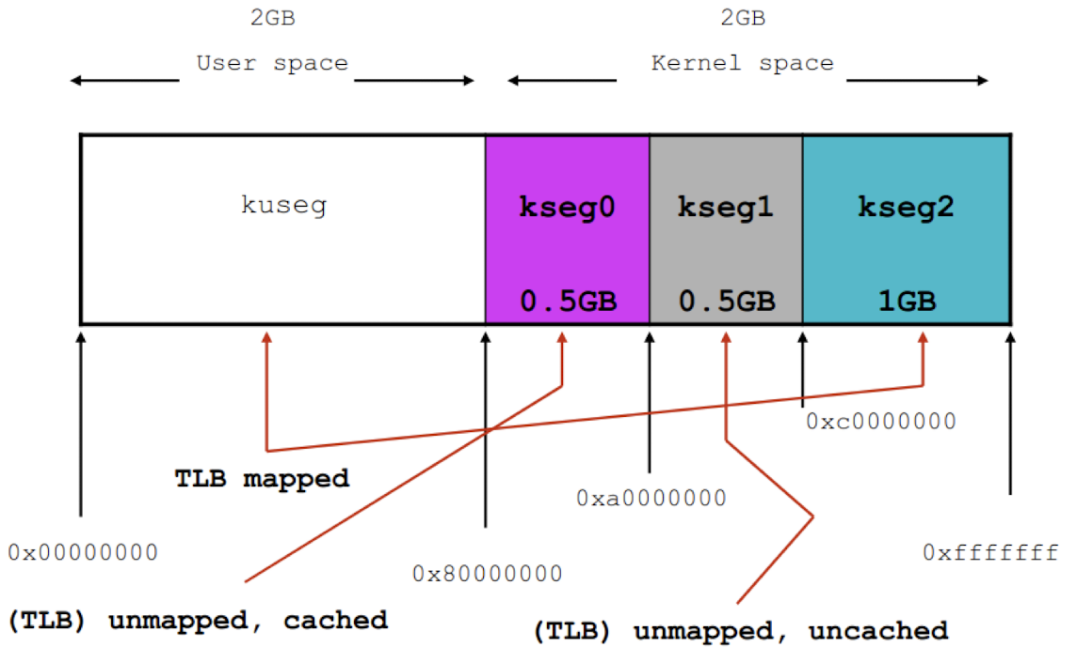


Figura 1.4: Lo spazio degli indirizzi utente di un processo, con il suo stack iniziale.

OS161

Differentemente in OS161 si abilita la traduzione logico-fisica per il kernel e si fa in modo che il kernel venga mappato in una porzione dello spazio di indirizzamento virtuale di ogni processo. In pratica questo vuol dire che, nello spazio di indirizzamento di un processo, alcuni indirizzi logici mapperanno su porzioni di memoria del processo utente mentre altri mapperanno su porzioni di memoria del kernel. Questa strategia implica che vi sia una strategia di protezione che non permetta ai processi utente di accedere alla parte del proprio spazio di indirizzamento virtuale contenente porzioni di kernel. Utilizzando questo approccio, il kernel ha un più facile accesso allo spazio d’indirizzamento del processo utente.

Il MIPS R3000 su cui gira OS161 ha indirizzi a 32 bit. Dei 4GB complessivi raggiungibili con indirizzi a 32 bit, 2GB sono dedicati allo user space e 2GB al kernel space.



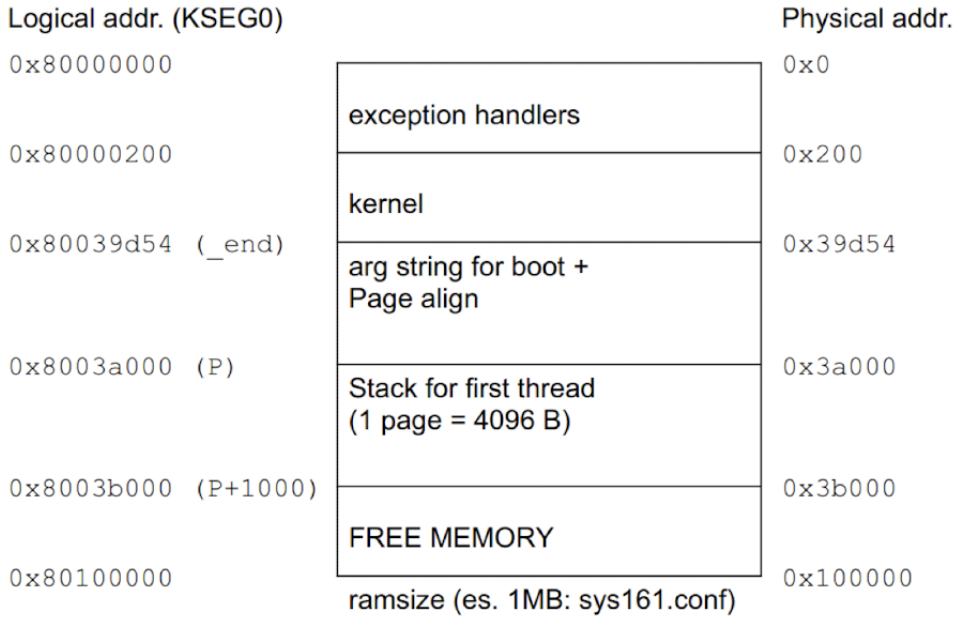
Come si vede dall'immagine, gli indirizzi riservati allo user space vanno da 0x00000000 a 0x80000000. Questo segmento di memoria prende il nome di kuseg e corrisponde alla parte bassa dello spazio di indirizzamento. I restanti 2GB da 0x80000000 a 0xffffffff sono riservati al kernel. Questo spazio è ulteriormente diviso in 3 parti da 0.5GB, 0.5GB e 1GB chiamate rispettivamente kseg0, kseg1 e kseg2. kseg0 non è mappato in TLB poiché non si prevede di fare paginazione su questa parte. kseg0 utilizza la cache. kseg1 allo stesso modo di kseg0 non è mappato in TLB, ma non utilizza neanche la cache. kseg2 è mappato in TLB: di conseguenza questa parte dell'address space virtuale del kernel è paginata.

In OS161 si utilizza soltanto una parte di quanto visto sull'architettura MIPS R3000. I processi utente risiederanno in kuseg, il codice ed i dati del kernel risiederanno in kseg0, i dispositivi di IO saranno accessibili in kseg1 e kseg2 non è usato. Da quanto detto si capisce che nessuna parte dello spazio degli indirizzi utilizzati nel kernel di os161l è paginata, a differenza di xv6.

Come detto, kseg2 non è utilizzato in OS161. Detto questo, la memoria virtuale di un processo può essere vista come composta da due parti: la parte degli indirizzi virtuali che iniziano con 0, corrispondente agli indirizzi logici utilizzabili dal processo utente (kuseg), e la parte degli indirizzi che iniziano per 1, corrispondente agli indirizzi logici del kernel (in kseg0 vi saranno codice e dati del kernel e kseg1 sarà riservato ai dispositivi di IO memory-mapped). Affinché un indirizzo che inizi con 0 possa essere tradotto, si ricorre dapprima al supporto hardware offerto dalla TLB (si ricordi che infatti kuseg è mappato in TLB): si proverà quindi ad identificare la entry corrispondente al numero di pagina estratto dall'indirizzo logico.

Il gestore di traduzione di OS161 prende il nome di DUMBVM, una cosa da tenere in considerazione in questo contesto è che DUMBVM fornisce un allocazione contigua delle pagine per un segmento. Nel caso in cui non si riesca ad identificare la entry corrispondente nella TLB, si dovrà ricorrere alla gestione del page fault fatta in kernel. Per quanto riguarda la traduzione degli indirizzi logici che iniziano con 1, non si utilizzerà il supporto offerto dalla TLB, ma si effettuerà una rilocazione andando a sottrarre all'indirizzo la quantità 0x80000000.

.



Al caricamento del kernel, nei primi 0x200 indirizzi fisici vanno a finire i gestori delle eccezioni, poi vi sarà il kernel, poi delle stringhe di boot. A seguito delle stringhe di boot c'è un allineamento per allineare l'indirizzo di partenza in modo tale che sia più facile gestire gli indirizzi a multipli di una pagina (0x3a000 infatti ha 13 bit a zero, ne servivano 12 per allineare a multipli di pagina).

System Calls

Esistono tre tipi di eventi che fanno sì che la CPU interrompa l'esecuzione ordinaria delle istruzioni e forzi un trasferimento del controllo a un codice speciale che gestisce l'evento. Una situazione è una chiamata di sistema, quando un programma utente esegue l'istruzione per chiedere al kernel di fare qualcosa per esso. Un'altra situazione è un'eccezione: un'istruzione (utente o kernel) fa qualcosa di illegale. La terza situazione è un'interruzione del dispositivo, quando un dispositivo segnala che necessita di attenzione, ad esempio quando l'hardware del disco termina una richiesta di lettura o scrittura.
Xv6 gestisce tutte le interruzioni (trap) nel kernel, non permettendo al codice utente di gestirle direttamente.

Esistono diverse ragioni che motivano questa scelta, le chiamate di sistema sono gestite nel kernel, quindi è naturale gestire anche le trap lì.
Gli interrupt, che provengono da dispositivi hardware, richiedono l'isolamento per garantire che solo il kernel possa accedervi. Xv6 risponde alle eccezioni (errori) terminando il programma che le ha causate, il che richiede una gestione nel kernel.

La gestione delle trap in Xv6 segue le seguenti fasi:

Inizialmente la CPU rileva l’interruzione o l’eccezione hardware, in seguito avviene il salvataggio dello stato del processo, infine il kernel gestisce la trap eseguendo l’azione appropriata come ad esempio eseguire una chiamata di sistema.

Sebbene i tre tipi di trap siano simili, Xv6 utilizza codice separato per gestire trap dallo spazio utente, trap dallo spazio kernel e interruzioni del timer. Questo offre maggiore chiarezza e controllo nella gestione di ciascun tipo di trap. Il codice che gestisce una trap è chiamato "gestore," e le prime istruzioni del gestore sono spesso scritte in assembler e talvolta chiamate "vettore" per indicare il punto di ingresso nella gestione della trap.

In xv6, un vincolo significativo nella progettazione della gestione delle trap è che l'hardware RISC-V non modifica le tabelle delle pagine quando genera una trap. Questo significa che l'indirizzo del gestore trap in stvec deve avere una mappatura valida nella tabella delle pagine utente, poiché questa tabella è in vigore quando inizia l'esecuzione del codice di gestione delle trap. Inoltre, il codice di gestione delle trap di xv6 deve passare alla tabella delle pagine del kernel. Affinché l'esecuzione possa continuare dopo questo passaggio, anche la tabella delle pagine del kernel deve avere una mappatura per il gestore puntato da stvec.

È importante notare che la CPU non effettua automaticamente questa transizione alla tabella delle pagine del kernel, non passa a uno stack nel kernel e non salva alcun registro diverso dal PC. Queste attività devono essere eseguite dal software del kernel.

Per affrontare questi requisiti, xv6 utilizza una "pagina trampolino". Questa pagina contiene "uservec", il codice di gestione delle trap di xv6 a cui stvec punta. La pagina trampolino è mappata nella tabella delle pagine di ogni processo all'indirizzo TRAMPOLINE, posizionato nella parte superiore dello spazio degli indirizzi virtuali. Questa collocazione è strategica poiché è sopra la memoria utilizzata dai programmi stessi. In questo modo, la pagina del trampolino assicura che ci siano mapping validi

sia nella tabella delle pagine utente che in quella del kernel, consentendo la corretta gestione delle trap nel contesto del sistema operativo.

Gestione delle Trap in user space e kernel space:

Punti Comuni:

1. Entrambe le gestioni delle trap (user space e kernel space) coinvolgono l'uso di una "trampoline page" mappata sia nello spazio utente che nel kernel.
2. In entrambi i casi, viene effettuato un salvataggio dei registri necessari per riprendere lo stato del processo interrotto in uno specifico contesto (trapframe) prima di gestire la trap.
3. Entrambe le gestioni delle trap coinvolgono la modifica del registro di controllo `stvec` per puntare al corretto gestore di trap.

Passaggi Differenti:

Gestione delle Trap in user space:

1. **uservec (kernel/trampoline.S:21):** Punto di ingresso iniziale per una trap dello spazio utente. Si occupa del salvataggio dei registri utente e prepara il contesto per ulteriori elaborazioni.
2. **usertrap (kernel/trap.c:37):** Gestisce la trap, cambia `stvec` se necessario, salva `sepc` (program counter utente) e chiama funzioni come `syscall` o `devintr`. Può anche gestire eccezioni o terminare un processo.
3. **usertrapret (kernel/trap.c:90):** Prepara i registri di controllo RISC-V per il ritorno allo spazio utente, cambia `stvec` a `uservec` e chiama `userret` sulla trampoline page.
4. **userret (kernel/trampoline.S:101):** Effettua la commutazione delle tabelle delle pagine e restituisce allo spazio utente, ripristinando i registri utente dal trapframe.

Gestione delle Trap in kernel space:

1. **kernelvec (kernel/kernelvec.S:12):** Punto di ingresso iniziale per una trap dal kernel. Salva tutti i 32 registri nello stack del thread del kernel interrotto.
2. **kerneltrap (kernel/trap.c:135):** Gestisce trap da interruzioni del dispositivo o eccezioni. Chiama `devintr` per le interruzioni del dispositivo e gestisce le eccezioni.
3. **Restituzione dopo la Trap nel Kernel:**
    - Salva `sepc` e lo stato corrente in `sstatus` quando inizia il lavoro.
    - Restituisce a `kernelvec` dopo aver completato il lavoro, ripristina `sepc` e lo stato e restituisce il controllo al kernel code interrotto.

In sintesi, mentre entrambe le gestioni coinvolgono l'uso della trampoline page e il salvataggio dei registri in un trapframe, i punti specifici di ingresso, le operazioni eseguite e il processo di ritorno differiscono tra lo spazio utente e lo spazio kernel. La gestione delle trap in user space è più focalizzata sulla gestione delle system call e delle eccezioni utente, mentre la gestione delle trap in kernel space si occupa principalmente delle interruzioni del dispositivo e delle eccezioni kernel, con la possibilità di eseguire `yield` in caso di timer interrupt.

OS161

In OS161 le system calls offrono ai processi un'interfaccia attraverso la quale accedere ad alcune funzioni del sistema operativo.

Il meccanismo di funzionamento è simile a quello appena descritto per xv6.

● Il primo evento che si verifica è una trap, e la prima fase consiste nel determinare se questa trap è stata causata da una richiesta di system call o da altri eventi come il timer, eccezioni, e così via.

● Successivamente, si esegue l'operazione di context-switching, che rappresenta il passaggio dalla modalità utente alla modalità kernel. Ciò comporta la creazione di un trap frame, salvato nello stack del kernel,che consente di riprendere lo stato del processo interrotto .

● avviene poi l'esecuzione vera e propria della system call (mappata con un numero identificativo che consente di individuare la syscall da gestire), che differisce a seconda di quale syscall è stata invocata;

● infine si ritorna alla user mode riprendendo l'esecuzione del programma;

Come in xv6, anche OS161 fornisce una lista di system calls a cui è associato un ID, che è il parametro che si utilizza per selezionare la system call da eseguire.

Sincronizzazione

I meccanismi di sincronizzazione rappresentano strumenti fondamentali per gestire il coordinamento tra processi, attività o dispositivi che operano in parallelo

.L'obiettivo primario è evitare problemi come "race condition" e "deadlock", preservando così la coerenza e la sicurezza del sistema.

La sezione di codice condivisa tra più processi è definita come critical section. Nel contesto di xv6 i principali meccanismi di sincronizzazione includono:

● ***Spinlock***: Utilizzato per ottenere l'accesso esclusivo e atomico a una risorsa condivisa da parte di processi o thread.

- Il meccanismo fa uso di busy waiting, ovvero attende in modo attivo il rilascio della risorsa.
- Definito tramite una struttura chiamata "spinlock" in spinlock.h, contenente informazioni come lo stato di acquisizione, il nome e il core della CPU associato.
- Per accedere in modo esclusivo a una specifica area di memoria, un thread deve acquisire il corrispondente spinlock e rilasciarlo al termine della sezione critica.
- Per evitare situazioni in cui più thread potrebbero acquisire simultaneamente lo spinlock, viene utilizzato un approccio basato sulla "test and set" a basso livello

  ○ ***acquire***: Funzione per acquisire il possesso dello spinlock, con un ciclo while che controlla continuamente lo stato dello spinlock.Quando una CPU acquisisce qualsiasi lock, xv6 disabilita sempre gli interrupt su quella CPU. Gli interrupt possono comunque verificarsi su altre CPU, consentendo a un gestore di interrupt di attendere che un thread rilasci uno spinlock, ma non sulla stessa CPU.

  ○ ***release***: Funzione per il rilascio dello spinlock.

  ○ ***initlock***: Funzione per definire i parametri dello spinlock, tra cui nome,stato di acquisizione e identificativo della CPU che lo possiede.

Queste azioni avvengono disabilitando e riabilitando gli interrupt, in modo da prevenire timeslicing quando il lock è già posseduto.

● ***Sleeplock***: è un meccanismo di sincronizzazione che implementa un approccio di attesa attiva (spinlock), insieme ad uno di attesa passiva (mettere il processo in stato di sleeping). Una volta acquisito lo spinlock della struttura dati si controlla se si può acquisire il possesso. Se questo è possibile si procede a impostare la variabile e a liberare lo spinlock, se invece non è possibile si mette il processo nello stato di sleeping. La caratteristica fondamentale è che i processi che vengono messi nello stato di sleeping possiedono lo spinlock, impedendo quindi race condition. I processi nello stato di sleeping vengono svegliati attraverso gli interrupt e per questo motivo hanno un utilizzo limitato. A differenza degli spinlock questo meccanismo è adatto

per le lunghe attese. Le funzioni per la gestione dello sleeplock sono simili a quelle dello spinlock e si trovano nel file *sleeplock.c*.

- Quando un processo chiama Sleep, esso viene collocato in una coda associata a una condizione specifica, identificata da una variabile chiamata void *chan. Questa coda rappresenta essenzialmente il luogo in cui i processi "addormentati" sono in attesa. Inoltre, per evitare possibili problemi quando più processi cercano di accedere alle stesse strutture dati, vengono utilizzati spinlock per garantire sezioni critiche brevi e controllate.

- Quando un'altra parte del sistema (un altro processo o componente) riconosce che la condizione è stata soddisfatta, chiama Wakeup specificando il canale (chan) in cui i processi sono in attesa. A questo punto, Wakeup agisce come la sveglia di una buona notte, spostando i processi dalla coda di attesa alla coda dei processi pronti. Ciò li rende eseguibili di nuovo e pronti a competere per il tempo della CPU.

OS161

- **Lock:**

  Lock è un meccanismo di sincronizzazione che assicura la mutua esclusione, consentendo a un solo thread di accedere alla sezione critica e implementando il concetto di possesso (ownership). È caratterizzato da un busy waiting molto limitato, rendendolo trascurabile rispetto alle spinlock.

- **Spinlock:**

  Spinlock, invece, è utilizzato per accedere alle sezioni critiche, garantire la mutua esclusione e eseguire istruzioni in modo atomico. Tuttavia, è associato a un approccio di busy-waiting, consumando cicli della CPU durante l'attesa. È possibile acquisire diverse spinlock, ma non è consentito acquisire più volte la stessa spinlock finché non è stata precedentemente rilasciata.

La principale differenza tra lock e spinlock risiede nel fatto che quest'ultima implica l'utilizzo di busy wait, mentre il lock no. Le spinlock sono adatte per operazioni molto rapide, ma meno per operazioni lunghe che consumano tempo di CPU. I lock, infatti, sono implementati con gli spinlock per gestire l'accesso a zone critiche contenenti poche istruzioni, mentre le operazioni più estese sono gestite dai lock.

- **Condition variable:**

  La condition variable è un meccanismo di sincronizzazione che permette di procedere nell'esecuzione del codice al verificarsi di una condizione soddisfatta da un thread diverso da quello corrente. Il thread viene messo in attesa in una coda, e quello che soddisfa la condizione decide se risvegliare tutti i thread in attesa o solo uno. Non impiega busy waiting, in quanto è costruita utilizzando i lock.

- **Semaphore:**

  Il semaforo permette l'accesso concorrente da parte di processi/thread a un numero limitato di risorse. Il programmatore decide quanti processi/thread possono accedere all'area protetta, impostando il semaforo. La decrementazione del contatore avviene quando un processo/thread cerca di accedere all'area protetta tramite la funzione wait(P). Il contatore viene incrementato quando il processo/thread ha terminato l'esecuzione del codice protetto, mediante la funzione signal(V). Le operazioni wait(P) e signal(V) sono eseguite in modo atomico. I semafori possono essere implementati con busy waiting tramite gli spinlock o con attesa passiva utilizzando Sleep/Wakeup.

- **Wchannel:**

  I wchannel sono simili alle condition variable, ma utilizzano gli spinlock (busy waiting) e sono utilizzabili solo nel kernel.

Per risvegliare un solo thread: cv_signal() per le condition variable e wake_one() per i wchannel. Per risvegliare più thread: cv_broadcast() per le condition variable e wchan_wakeall() per i wchannel. Per mettere in attesa i thread in attesa di segnali: wchan_sleep() per i wchannel e cv_wait() per le condition variable.

IMPLEMENTAZIONE DI NUOVE FUNZIONALITÀ

Syscall ps:

la syscall restituisce la lista di processi attivi con relativi pid, la syscall è stata implementata esportando la tabella dei processi.

Per realizzare una system call in **xv6**:

- *syscall.h* contiene una mappatura tra il nome della system call e il numero con cui viene identificata. È necessario aggiungere a queste mappature la nuova system call, assegnando a questa (*SYS_name*) un intero positivo diverso da quelli già usati.

- *syscall.c* contiene funzioni di aiuto per analizzare gli argomenti delle system call e puntatori alle implementazioni delle system call. In corrispondenza di (*syscalls[]),
  bisogna associare alla costante definita nel punto precedente la funzione che la esegue, denominata *sys_name*.
  Inoltre bisogna renderla disponibile all'esterno di tale file sorgente, quindi prima la si riscrive anticipando a questa la voce "extern" e scrivendo il tipo di parametri ed il tipo di ritorno.

- *sysproc.c* contiene le implementazioni delle system call relative ai processi. Qui si aggiungerà il codice delle system call *sys_name()* che ritorna un intero

- in *proc.c*, definire nel file sorgente citato nel punto precedente la funzione *name()* e riportare il suo prototipo nei file header *defs.h* (file header al quale si riferisce *proc.c*) e *user.h* (file header al quale si riferisce *sysproc.c*) dove sono definite le altre funzioni per le system call.

- *usys.S* contiene un elenco di system call esportate dal kernel. Al suo interno bisogna aggiungere una voce *SYSCALL(name)*, dove al posto di *name* si riporta il nome della funzione precedentemente definita in *proc.c*.

- affinché tale system call possa essere invocata da linea di comando, bisogna creare un apposito file sorgente *name.c* contenente la funzione main e che invochi tale funzione. Affinché *name.c* possa invocarla questo deve includere la libreria *user.h*.
- in *Makefile*, si riporta il nome del file sorgente affinché possa essere richiamato da linea di comando: si aggiunge il suo nome alla voce *UPROGS*, formattata come gli altri file.
- a questo punto si compila e si esegue il kernel con i soliti comandi (make clean - make qemu): si può notare, digitando il comando "ls", che nella lista dei comandi/eseguibili è presente la funzione implementata.

Dimensione massima di un singolo file:

La seconda funzionalità implementata è l'incremento della dimensione massima di un file all'interno del sistema opertivo.
Inizialmente i file in xv6 sono limitati a a 268 blocchi questo limite deriva dal fatto che un inode xv6 contiene 12 blocchi diretti e un numero di blocco indiretto singolo, che punta a un blocco che contiene fino a 256 blocchi, per un totale di 12+256=268

Lo scopo di questa implementazione è quello di incrementare il limite appena citato andando ad inserire un blocco indiretto doppio, in questo modo un inode potrà contenere: $11+256+256^2=65803$ blocchi.

Andando a selezionare il test writebig attraverso il comando “writebigfile”, oppure tramite “usertests writebig” viene creato il file più grande possibile e segnala tale dimensione.

- fs.h contiene le informazioni relative agli inode, per implementare la doppia indirezione abbiamo effettuato le seguenti modifiche:
    - NDIRECT = 11
      Un blocco diretto verrà utilizzato per la doppia indirezione (si passa da 12 a 11)

    - #define NDOUBLYI_NDIRECT NINDIRECT*NINDIRECT
    - #define MAXFILE (NDIRECT + NINDIRECT+NDOUBLYI_NDIRECT)
    - addrs[] passa da uint addrs[NDIRECT+1] a uint addrs[NDIRECT+2];

- fs.c contiene l’implementazione del filesystem.
    - nella funzione bmap() sono state modificate le condizioni per supportare un blocco indiretto doppio:

    - **Calcolo dell'indice nel blocco indiretto doppio:**
        - bn è l'indice del blocco logico che si sta cercando.
        - Si verifica se bn è nell'intervallo dei blocchi diretti, indiretti e indiretti doppi.
        - Se bn è nei blocchi indiretti doppi, l'indice viene adeguato.
    - **Calcolo degli indici dei blocchi:**
        - `double_index` rappresenta l'indice del blocco indiretto doppio.
        - `single_index` rappresenta l'indice del blocco indiretto singolo.
    - **Caricamento del blocco indiretto doppio:**
        - Si carica il blocco indiretto doppio di primo livello, identificato da `ip->addrs[NDIRECT + 1]`.
        - Se il blocco indiretto doppio non è ancora allocato, viene allocato con `balloc`.
    - **Caricamento del blocco indiretto singolo dal blocco indiretto doppio:**
        - Si carica il blocco indiretto singolo di secondo livello dal blocco indiretto doppio.
        - Se il blocco indiretto singolo non è ancora allocato, viene allocato con `balloc`.
    - **Caricamento del blocco diretto dal blocco indiretto singolo:**

        - Si carica il blocco diretto di primo livello dal blocco indiretto singolo.
        - Se il blocco diretto non è ancora allocato, viene allocato con `balloc`.
    - **Ritorno dell'indirizzo del blocco dati:**
        - L'indirizzo del blocco dati viene calcolato usando l'indice `pos` all'interno del blocco indiretto singolo.
        - Se il blocco dati non è ancora allocato, viene allocato con `balloc`.

- itrunc() è la funzione usata per liberare la memoria, per adattarsi alla doppia indirezione sono state effettuate le seguenti modifiche:

    - **Ricavo inode di indirezione doppia di primo livello:**
        - ip->addrs[NDIRECT]
    - **Ricavo gli inodes di indirezione doppia di secondo livello:**
        - for (i = 0; i < NINDIRECT; i++)
    - **Libero i blocchi dati :**
        - for (int j = 0; j < NINDIRECT; j++) {
                if (a2[j]) {
                bfree(ip->dev, a2[j]); } }

```
// Free the doubly indirect block
    if (ip->addrs[NDIRECT]) {
        bp = bread(ip->dev, ip->addrs[NDIRECT]);
        uint *a = (uint*)bp->data;
        for (i = 0; i < NINDIRECT; i++) {
            if (a[i]) {
                struct buf *bp2 = bread(ip->dev, a[i]);
                uint *a2 = (uint*)bp2->data;
                for (int j = 0; j < NINDIRECT; j++) {
                    if (a2[j]) {
                        bfree(ip->dev, a2[j]);
                    }
                }
                brelse(bp2);
                bfree(ip->dev, a[i]);
            }
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT]);
        ip->addrs[NDIRECT] = 0;
    }
```

```
    // CODICE DALLA RIGA 418 bmap()

else if (bn < NDIRECT + NINDIRECT + NDOUBLY_INDIRECT) {
        // Doubly indirect block logic
        bn -= NDIRECT + NINDIRECT; // Adjust bn for doubly indirect
        uint double_index = bn / (NINDIRECT * NINDIRECT);
        uint single_index = (bn / NINDIRECT) % NINDIRECT;
        // Load the first-level doubly indirect block
        uint addr_double = ip->addrs[NDIRECT + 1];
        if (addr_double == 0) {
            addr_double = balloc(ip->dev);
            if (addr_double == 0) {
                return 0; // Error handling}
            ip->addrs[NDIRECT + 1] = addr_double;}
        // Load the second-level indirect block from the first-level doubly indirect block
        bp = bread(ip->dev, addr_double);
        a = (uint*)bp->data;
        uint addr_indirect = a[double_index];
        if (addr_indirect == 0) {
            addr_indirect = balloc(ip->dev);
            if (addr_indirect == 0) {
                brelse(bp);
                return 0;}  // Error handling
            a[double_index] = addr_indirect;
            log_write(bp);}
        brelse(bp);
        // Now load the first-level indirect block from the second-level indirect block
        bp = bread(ip->dev, addr_indirect);
        a = (uint*)bp->data;
        uint addr_single = a[single_index];
        if (addr_single == 0) {
            addr_single = balloc(ip->dev);
            if (addr_single == 0) {
                brelse(bp);
                return 0; // Error handling }
            a[single_index] = addr_single;
            log_write(bp); }
        brelse(bp);
        // Finally, load the data block from the first-level indirect block
        bp = bread(ip->dev, addr_single);
        a = (uint*)bp->data;
        uint pos = bn % NINDIRECT;
        addr = a[pos];
        if (addr == 0) {
            addr = balloc(ip->dev);
            if (addr == 0) {
                brelse(bp);
                return 0; // Error handling}
            a[pos] = addr;
            log_write(bp);}
        brelse(bp);
        return addr;
    } else {
        // Out of range
        panic("bmap: out of range");}
```