Il laboratorio ci ha richiesto di costruire un log con date e messaggi inseriti da più utenti. Per via del funzionamento dei lock potrebbe succedere che con più scritture contemporaneamente, dopo che il primo ha scritto il suo messaggio la seconda e la terza sono in attesa da un po’ sebbene abbiano già preso l’ora: poiché non è garantito che il lock come secondo lo prenda il secondo, ma magari il terzo, può succedere che le date non siano in ordine strettamente crescente. La proposta di questa parte facoltativa è quindi di aggiungere un secondo timestamp dopo il messaggio che evidenzi l’istante in cui è stato preso il lock di scrittura. L’unica piccola difficoltà è che i record vengono inseriti in un buffer di dimensioni predefinite, quindi per evitare di uscire dall’area di memoria allocata bisogna effettuare un troncamento dei caratteri che andrebbero a debordare. Bisogna scrivere due righe di relazione sul funzionamento / malfunzionamento del programma.

Oggi parliamo di una cosa a metà tra software e hardware, ossia il sistema operativo.

Ripartiamo dal principio di Denning della separazione e virtualizzazione. Abbiamo gli utenti da una parte, il nucleo al centro e le risorse dall’altra. È il nucleo a fare da intermediario e da filtro, secondo una politica che è stata definita in anticipo. Il permesso di scrittura potrebbe essere suddiviso in poter scrivere un file da zero o poter solo aggiungere informazioni.

Come fa il nucleo a realizzare la virtualizzazione delle risorse? (dove tali risorse sono le unità a disco, le stampanti, le tastiere ecc.). Nel passato sono state sviluppate tre tipi di organizzazioni per il kernel del sistema operativo. Un modo che è stato usato inizialmente era quello di avere dei Kernel monolitici: chi scrive il nucleo si preoccupa di scriverlo per un particolare sistema operativo e perché possa gestire solo un certo numero specifico di risorse. Il problema del kernel monolitico è che va ricompilato ogni qualvolta cambiano la quantità e il tipo di risorse del calcolatore. L’approccio opposto è il microkernel, in questo caso anziché avere un unico programmone che gestisce tutte le risorse, si riduce al minimo il nucleo di sicurezza (e il minimo che serve è quello che fa la virtualizzazione) e riduce tutto il resto a una parte applicativa che lavora in modalità non privilegiata. A livello hardware questo richiede un MMU, un meccanismo di Interrupt e Trap e un meccanismo di Istruzioni privilegiate. In questo modo possiamo differenziare l’esecuzione di codice da parte del Kernel (che lavora in modalità privilegiata) da quella da parte di un’applicazione.

Se il kernel monolitico rappresenta la ridondanza di controllo, il microkernel rappresenta l’economia di controllo.

Le applicazioni utilizzate dal microkernel per interagire poi con le risorse non sono applicazioni normali e vengono chiamate server, tuttavia funzionano col processore in modalità non privilegiata.

Ad esempio si dà a un server la possibilità di scrivere sui registri della RAM, questo sembrerebbe di violare il primo principio di Denning, però tale server non è un’applicazione al servizio dell’utente, ma è surrogata al microkernel e dunque è parte del sistema operativo. Il microkernel infatti manda dei messaggi di richiesta al server giusto e questo farà da intermediario tra applicazione e risorsa. Quali sono i vantaggi e gli svantaggi di un’organizzazione di tipo microkernel?  
Il vantaggio più grande è avere solo una piccola parte del sistema operativo che funziona in modalità non privilegiata, mentre il resto funziona in modalità non privilegiata (che rende più facile programmare il sistema operativo perché in modalità non privilegiata abbiamo il meccanismo delle trap, che ci permette di non causare danni gravi se ci sono dei bug). La cosa non si applica al sistema monolitico, un solo baco lì può fare danni su tutto il sistema (mentre col microkernel se c’è un bug nel server della gestione dell’unità a disco questo causerà problemi solo lì). Il microkernel è inoltre più facile da debuggare, perché ha meno linee di codice raggruppate in un solo posto. Un altro vantaggio molto importante è che siccome il microkernel si occupa solo della gestione del processore, in qualsiasi momento si può mandare in esecuzione un nuovo server per gestire una nuova risorsa: rende molto più facile effettuare un upgrade e dà la possibilità di avere dispositivi rimuovibili (come una chiavetta usb, inutilizzabile in un nucleo monolitico).

La maggioranza delle implementazioni sono una via di mezzo tra le due implementazioni. Tale kernel è detto Kernel modulare. In esso c’è una parte fissa che corrisponde grossomodo a un microkernel, poi sono previsti dei moduli per la gestione delle risorse: questi moduli possono essere inclusi o tolti dinamicamente dal sistema attraverso un’operazione di linking del sistema. La differenza tra moduli e server è che i moduli lavorano sempre in modalità privilegiata. Si ha così la gestione dinamica del sistema che permette la sostituzione e la rimozione di dispositivi senza dover effettuare la ricompilazione del kernel. Rispetto all’organizzazione microkernel lo svantaggio è che i moduli (anche detti Device Driver) lavorano in modalità privilegiata: se c’è un bug i danni sul sistema possono essere devastanti (perché i moduli non sono confinati), inoltre i device driver possono essere dei veicoli di inserzione di malware all’interno del sistema. Un aspetto positivo però è quello di mantenere la configurazione in cui gli utenti devono sempre e solo contattare il kernel per l’accesso alle risorse e non sono costretti a passare per dei server. Il passaggio da un kernel monolitico e quello modulare è più tranquillo rispetto il passaggio da microkernel a monolitico (questo è anche il motivo per cui i microkernel non hanno mai ottenuto un vero successo commerciale).

Passando a qualcosa di più concreto, vediamo i sistemi POSIX (la generalizzazione di Unix). Una delle caratteristiche principali di Unix è stato di semplificare notevolmente l’interfaccia di programmazione delle applicazioni e del sistema operativo, basando tutto sull’idea di “File”. Più che a livello di file, da un punto di vista tecnico, si tratta in realtà di stream di byte (byte stream). L’idea è avere la possibilità di creare sequenze di byte che contengono informazioni arbitrarie e a uno stream abbiamo la possibilità di aggiungere una quantità arbitraria di byte (possiamo avere uno stream di 10 byte e aggiungervene altri 3). I byte sono inseriti in ordine di inserzione (la struttura è ordinata e i byte possono essere numerati). Dopo aver scritto uno stream c’è la possibilità di accedere a quello stream attraverso un’operazione di lettura. Si possono leggere quanti bye si vuole dello stream (per esempio prima 2, poi 5, poi 6): l’unica cosa da sottolineare è che non si può leggere più byte di quanti ne sono presenti nello stream.   
Esempi pratici potrebbero essere le stringhe di caratteri. Non sono solo i file a usare quel concetto di stream, per esempio potrebbero essere anche i socket, o le pipe a usarli.   
Questa semplificazione si trova in tutti i sistemi moderni. Vediamo cosa comporta l’implementazione a livello hardware di quella cosa (ad esempio di un device driver).

Esempio: prendiamo un dispositivo che sia di tipo stampante, quindi accetti come unica operazione la scrittura. Un’applicazione può mandare per esempio una stringa di caratteri alla stampante e ottenere che tale stringa venga stampata. Per gestire il dispositivo stampante il concetto fondamentale è di utilizzare dei buffer. Questi buffer sono aree di memoria che possono contenere una quantità sufficiente di byte, così da poter contenere il byte stream. Lo schema è quindi del tipo:

Applicazione: richiama una system call di tipo WRITE alla quale passa come parametro l’indirizzo RAM della stringa da stampare (nel processore Amber potremmo usare la SWI) e la sua lunghezza.

Sistema: viene copiata tale stringa in un buffer

Stampante: Presenta un registro memory-mapped capace di contenere un byte. In questo viene inserito dal sistema il primo byte della stringa e inizia la stampa (operazione durante la quale il sistema fa altre cose).

Quando il primo carattere viene stampato viene inviato dalla stampante un flag di interrupt e viene copiato un altro byte nel registro. Si prosegue fino a che non si è copiato l’ultimo byte del buffer, quando ciò avviene il buffer diventa “libero” (non vengono cambiati i dati ma diventa sovrascrivibile).

Ovviamente la WRITE non è l’unica system call che si comporta in questo modo, ci sono anche la putchar, la printf, la fprintf ecc. Queste funzionalità però, sono implementate all’interno di una liberia standard (la glibc), che utilizza nuovamente al suo interno la stessa idea dei buffer per poi alla fine far intervenire il sistema operativo su una chiamata di tipo WRITE: dunque queste funzioni sono un ulteriore livello di astrazione (e la nostra applicazione si comporterà normalmente).

L’idea dello stream viene implementata attraverso l’uso delle funzioni standard della libreria, che permettono di aggiungere nuovi caratteri allo standard output (che è lo stream). Questo permette di limitare l’attività dei device driver.

Per poter svuotare il buffer dai caratteri già stampati e ormai inutili si utilizza l’operazione di flush: c’è una system call di fflush a cui si passa un file descriptor che indica il buffer da pulire. Alternativamente se si sta utilizzando la printf, inserire il carattere di fine riga ‘\n’ questo provoca automaticamente la chiamata dell’operazione di flush. In mancanza del flush la stringa può rimanere nel buffer anche a fine della stampa. Oltretutto se non si effettua il flush il contenuto del buffer potrebbe non venire stampato in primo luogo (e quindi potrebbe essere incerto il punto di terminazione del programma).

Quand’è che le informazioni passano dal buffer applicativo al buffer del sistema operativo? Quando si chiama la flush: ciò avviene quando si riempie lo stream, quando si chiama la fflush con file descriptor che descrive quel buffer o quando (in caso di stringhe formattate) si inserisce il carattere di newline.