Si consideri un insieme di 4 processi {P1, P2, P3, P4} ciascuno dei quali scrive periodicamente un nuovo messaggio su uno slot di una memoria condivisa M. Il processo Pi scrive esclusivamente sul corrispettivo slot M[i] della memoria condivisa. Un ulteriore processo REPLY legge periodicamente in ordine circolare i messaggi scritti dai processi Pi. Ogni 2 nuovi messaggi letti, REPLY scrive una risposta su una memoria condivisa R, e la risposta deve essere letta dai 2 processi Pi mittenti dei messaggi letti da REPLY. La lettura di REPLY dagli slot della memoria condivisa M è bloccante in assenza di nuovi messaggi, così come la lettura della risposta da parte dei processi Pi su R in assenza di risposta, mentre le scritture sugli slot di M sono concorrenti. Si schematizzi la soluzione del suddetto problema di sincronizzazione, usando solo semafori, fornendo lo pseudo-codice delle procedure SCRIVI-LEGGI usata dai processi Pi e LEGGI-RISPONDI usata dal processo REPLY.

Mi servono i seguenti semafori:

1) Writers-P: {4 entry, 1 token l'una} -> a me non interessa l'ordine di scrittura, ma devo assicurare che le informazioni non vengano sovrapposte; tutte le entry sono inizialmente poste ad 1;
2) Reader: {4 entry, 1 token l'una} -> mi interessa un ordine di lettura circolare;
3) Writer-R: {1 entry, 2 token}; inizialmente la entry dispone di 2 token;
4) Readers-P: {4 entry, 1 token l'una} -> devo segnalare capillarmente chi può leggere ora e chi no;

// Pseudocodice:
SCRIVI LEGGI: // AGGIUNGI WHILE 1:

```
WAIT(Writers-P, entry i-esima);
<Scrivi sulla tua frazione di memoria condivisa>;
SIGNAL(Reader, entry i-esima);

WAIT(Readers-P, entry i-esima);
<Leggi il messaggio depositato su R>;
SIGNAL(Write-R);
```

LEGGI – RISPONDI:

```
i= 0;
// Sezione critica di codice:
while(1){
        WAIT(Reader, entry i-esima);
        <Leggi informazioni dal frammento di memoria i-esimo>;
        if(i== 1 || i == 3){
                WAIT(Writer-R, 2);
                <Scrivi un messaggio sulla memoria condivisa R>;
                SIGNAL(Readers-P, entry i-1esima);
                SIGNAL(Readers-P, entry i-esima);
        }
        SIGNAL(Writers-P, i);
        i= (i == 4 ? 0 : i+1);
}
```

Domanda 1 (5.25 punti)
Si descriva lo scheduler di CPU Unix tradizionale, indicando anche quali system-call possono essere utilizzate per configurarlo.

Lo Scheduler UNIX nacque per macchine single-processor, ed era ovviamente improntato ai processi. La successiva evoluzione dell'informatica ha decretato un suo sviluppo in una forma tutt'oggi utilizzata anche per i thread, sebbene noi vedremo in questa sede la versione per processi.

Si riprende il concetto di più code di priorità già introdotto in multi-level-feedback-queuing. In questo caso, tuttavia, ogni coda è associata ad un livello di priorità, che va dalla più bassa (20, usualmente utilizzata da processi CPU bound) alla più alta (-19, generalmente a carico degli I/O bound); all'interno della singola coda l'algoritmo che gestisce l'entrata in CPU è di tipo Round Robin. Il criterio con il quale si decide in quale coda sia da destinare un processo è dato dalla seguente equazione:

$$P = Base + \frac{CPU}{2} + niceness, dove, generalmente, Base\ e\ Niceness\ sono\ posti\ a\ zero.$$

Tuttavia, per rendere più o meno "competitivo" un processo è possibile variare la sua niceness. E' possibile alzare la Niceness verso valori positivi (N>0 -> P crescente) senza alcun problema: infatti, in questo modo il processo è più "amichevole" verso gli altri e così diverrà meno competitivo. Sono necessari invece dei permessi particolari per abbassare la niceness sotto lo zero ed avvicinarsi sempre più alla priorità di -19. Si può evincere la priorità di un processo mediante la syscall getpriority, mentre si può settare la priority con setpriority. Per modificare la niceness, invece, si utilizza la syscall nice(int value), dove value appartiene a [-20, 19].

Per un contesto multithreading, orientato quindi a macchine multiprocessore si può introdurre anche il concetto di affinità con il singolo processore, ma esime dalla trattazione richiesta.

Domanda 3 (5.25 punti)

Descrivere la tecnica di gestione della memoria basata su partizioni dinamiche. Si consideri inoltre una sequenza di generazione di 4 processi, P4, P1, P2 e P3. P1 e P2 hanno taglia 1MB, P3 ha taglia 2 MB e P4 ha taglia 4 MB. Si consideri inoltre una memoria di lavoro di 7 MB di cui 1 MB sia riservato per il sistema operativo. Si determini quale deve essere la relazione tra il tempo di completamento di P1 e P2 ed il tempo di nascita di P3 affinchè, e sotto quali condizioni, ognuno dei 4 processi posssa essere correttamente allocato in memoria all'atto della sua creazione.

Si usano partizioni di numero e dimensione variabile; quando un processo entra in memoria, gli viene allocata memoria pari alla sua dimensione e mai di più. Inizialmente abbiamo quindi una partizione occupata dal SO pari a 1MB e una partizione libera di 6MB, occupata successivamente da P4 di taglia 4MB, ergo ora lo spazio è diviso in 1MB->occupata, 4MB->occupata, 2MB->libera. In tale contesto, i processi P1, P2 possono entrare senza problemi e saturare lo spazio di memoria. La soluzione ottimale risiede nel completamento di P1 e P2 prima della generazione di P3. In sostanza, se P3 dall'istante in cui entra in CPU P2 ci mette un tempo pari ad h, P1 non può superare h + tempo generazione P2, e P2 non può superare h. Un'altra possibilità sarebbe quella di fare swap out di P4, ma questo porterebbe a frammentazione esterna della memoria, eventualità che si vuole evitare a causa dell'eccessivo costo della ricompattazione.

Domanda 2 (5.25 punti)

Si descriva il metodo di allocazione dei file indicizzato. Inoltre, si consideri un dispositivo di memoria di massa con blocchi di taglia pari a 4 KB, indici di blocchi di taglia pari a 4 byte, ed un record di sistema contenente in totale 12 indici di cui N diretti ed M indiretti. Si determini il valore di N ed M, qualora esista, che possa permettere di allocare file di taglia almeno pari a 4 GB.

Il numero di indirizzi possibili è dato da: $\frac{(Dimensione\ blocco)}{Dimensione\ indici\ di\ blocco} = \frac{(4*1024)}{4} = 1024$ possibili indirizzi. Il nostro target è quello di ottenere allocazione di file grandi almeno $2^{32} (ovvero, 4 * 2^{30}, dove\ 2^{30} significa\ 1GB)$, quindi bisogna semplicemente andare a tentativi e vedere quanto la dimensione è maggiore di 2^32.

Siccome il testo non autorizza l'utilizzo di indici indiretti di livello superiore al primo (e non possiamo fare alcuna ipotesi sostanziale al fine di usare le extents) l'ipotesi migliore che possiamo fare è che tutti gli indici siano indiretti di primo livello, garantendo quindi ognuno 1024 possibili indirizzamenti. Tuttavia, non è abbastanza per caricare un file così massivo in memoria. Un rapido calcolo ce lo conferma:

$$Dimensione = (numero\ di\ indici) * Numero\ di\ entry * dimensione\ in\ memoria\ di\ un\ singolo\ blocco$$
$$= 1024 * 12 * 1024 * 4 = 48 * 1024^2 = ordine\ dei\ MegaByte$$

Ergo, non è possibile allocare questo file in memoria. Nota: se fosse stata possibile l'indicizzazione a più livelli indiretti, ci sarebbe bastato un solo indice di secondo livello indiretto ed uno di primo livello indiretto, più 10 indici diretti. Infatti:

$$Dimensione = (1024 + 1024^2 + 10) * 1024 * 4\ che\ è\ di\ poco\ superiore\ a\ 4\ GB.$$

Per usare le extents ci sarebbero volute invece ipotesi diverse, come il numero di blocchi in sequenza disponibili, ed anche la famiglia di Sistemi considerata (esempio, nei moderni sistemi UNIX usiamo le Ext4).

Domanda 4 (5.25 punti)
Si consideri un insieme di N processi {P0, P2, P3, …., PN-1} che scambiano messaggi circolarmente. Lo scambio viene attivato dall'inserimento di un messaggio da parte di P0 all'interno di un'area di memoria condivisa M. Non appena un messaggio viene inserito da un processo Pi nella memoria condivisa, Pj (con j = i+1%N) può leggerlo e può produrre un ulteriore messaggio da inserire nella memoria condivisa M. La lettura della memoria condivisa M è bloccante in assenza di nuovi messaggi. Si schematizzi la soluzione del suddetto problema di sincronizzazione, usando solo semafori, fornendo lo pseudo-codice delle procedure SCRIVI e LEGGI usate da ciascuno dei procssi Pi. Si noti che le procedure sono 2 distinte per ogni processo poichè la lettura da M da parte di Pi non è necessariamente contestuale alla produzione di un nuovo messaggio da scrivere in M.

Ho bisogno dei seguenti semafori:

1) Writers: {N entry, 1 token l'una} -> Sono interessato ad un ordine di scrittura circolare; inizialmente solamente il processo P0 sarà provvisto di un token, mentre gli altri non hanno token; nota: il semaforo ha il solo scopo di rendere sequenziale l'accesso alla memoria, perché per come è strutturato il codice (lettura bloccante) non vi può essere competizione per la risorsa;
2) Readers: {N entry, 1 token l'una} -> Sono interessato ad ordine di lettura lineare; inoltre, tutte le entry sono inizialmente poste a zero;

SCRIVI:

```
WAIT(Writers, entry i-esima);
<Deposita l'informazione sulla memoria condivisa>;
j= (i == N -1 ? 0 : i+1);
SIGNAL(Readers, entry j-esima); // In questo modo avverto il processo a seguire che ci sono nuove informazioni;
SIGNAL(Writers, entry j-esima); // In questo modo permetto al processo a seguire di scrivere;
```

LEGGI:

```
WAIT(Readers, entry i-esima);
<Leggi l'informazione dalla memoria condivisa>;
SCRIVI;
```

Domanda 3 (5.25 punti)
Descrivere la struttura e l'utilizzo della tabella delle pagine e la relazione tra questa ed i componenti hardware presenti su una architettura di processore convenzionale. Considerando inoltre uno schema di paginazione a 3 livelli in cui la tabella di primo livello sia costituita da 4 K elementi, quella di secondo livello da 2 K elementi e quella di terzo livello da 1 K elementi, si determini il numero massimo di pagine gestibili all'interno dello spazio di indirizzamento di un processo.

Nella sua forma più generale, la Tabella delle pagine prevede due bit: presence e dirty bit, affiancati ad una pagina. Il primo indica se vi è una pagina mappata nella memoria di lavoro, mentre il secondo indica se tale pagina deve essere ricopiata su hard disk. Per rinvenire una pagina di memoria, si fornisce un indirizzo ad n bit, di cui i primi k bit sono utilizzati per indicare il numero della pagina, mentre gli ultimi r bit sono utilizzati per un offset interno alla pagina, per quindi rinvenire il dato richiesto.

La tabella delle pagine è propria di ogni processo e caricata in memoria di lavoro (leggasi: possiede un indirizzo fisico, ed è necessaria quindi la traduzione a indirizzo logico mediante componenti hardware quali la MMU). Il Sistema Operativo mantiene un registro che sappia puntare effettivamente alla tabella delle pagine per il processo corrente. Questo tuttavia è uno svantaggio: infatti, questo implica che devo fare accessi potenzialmente costosi in memoria di lavoro. Per ovviare a questo problema, è stato aggiunto un "acceleratore" hardware per il binding degli indirizzi detto TLB. Il TLB alla fine altro non è che un'area di memoria dove vengono mantenute associazioni tra numero di pagina e relativo frame per il processo attuale (leggasi: ad un cambio di contesto, bisogna flusharlo, in quanto altrimenti il prossimo processo magari trova il numero giusto di pagina, ma viene portato al frame del processo precedente: disastroso). Il problema è quando vi è una miss nel TLB, perché in questo caso bisogna necessariamente accedere alla tabella delle pagine. Capire se vi è stato o meno un miss sul TLB è possibile grazie a del microcodice della CPU, ed è quindi palese che la struttura della tabella dipenda dalla logica e dal microcodice proprietari del processore in questione.

Per quanto riguarda il problema da risolvere, è come applicare la memoria virtuale alla memoria virtuale, ovvero astrarre ulteriormente la tabella delle pagine. La tabella di primo livello possiede 4k entry che portano (ognuna di loro) a tabelle di secondo livello. Ognuna di queste tabelle possiede a sua volta 2K elementi che puntano a tabelle di terzo livello da 1k elementi.

Quindi basta risolvere: $4K * 2K * 1K = 8 * 10^9$.

Domanda 1 (5.25 punti)

Si descriva lo scheduler di CPU Windows, includendo nella descrizione anche le system-call utilizzabili per configurarlo. Si consideri inoltre uno scenario ideale in cui 2 thread T1 e T2 appartenenti ad uno stesso processo, ed aventi tutti e due la stessa priorità base minore di 15, vengano generati allo stesso istante di tempo T0. Il thread T1 è puramente CPU bound e richiede 100 msec di tempo di CPU. Il thread T2 è I/O bound e necessita di eseguire 50 CPU burst ciascuno di durata pari a 10 msec. Supponendo che il dispositivo con cui interagisce T2 abbia una latenza di operazione pari a 10 msec, che il quanto di tempo utilizzato dallo scheduler Windows sia pari a 10 msec, che le operazioni eseguite dal sistema operativo per la gestione dei thread e del loro scheduling abbiano latenza nulla, che sia disponibile una sola CPU, e che non vi siano altri thread da gestire, si determini il tempo di completamento di ciascuno dei due thread (T1 e T2) ed il valore della loro priorita' all'atto della loro ultima schedulazione in CPU.

Lo Scheduler di CPU Windows riprende molti concetti già introdotti in SVR3 e in Multi-level-feedback-queuing. Ovviamente, è necessario tenere in considerazione che Windows nasce come sistema multi-threading, e ad essere schedulati ed eseguiti dalle CPU sono, ovviamente, i threads. Come in VSR3, ritroviamo delle code di priorità; all'interno della singola coda, la gestione è (come in VSR3) di tipo Round Robin. Ciò che cambia è la gestione delle priorità. In Windows troviamo infatti due fasce di priorità, la prima è detta *variable* [0;15], mentre la seconda è detta *real-time* [16,31]. Ovviamente, il livello più prioritario è il 31, ed il meno prioritario è lo 0. Come in VSR3 è possibile spostarsi all'interno delle varie code di priorità, ma sotto determinate condizioni:

1) I threads nella fascia *real-time* sono destinati a rimanere per sempre nella coda cui erano stati originariamente destinati, ergo non è possibile cambiare la loro priorità con la syscall *SetThreadPriority*;
2) Al processo generatore è associata una priorità di base. Tale priorità non è volta alla schedulazione del processo (Windows schedula ed esegue solamente threads), quanto fornire un vincolo di movimento per i threads generati dal processo; infatti, i processi nella fascia *variable* possono muoversi da 0 a 15, ma questo è possibile solamente in un range deciso dalla priorità del processo padre. Tale priorità del processo padre può essere modificata mediante la syscall SetPriorityClass;
3) La Priorità di un thread può cambiare (sullo spunto di ciò che accadeva in Multi-level-feedback-queuing) in base al tempo di utilizzo della CPU. Se il thread non utilizza tutto il quanto di tempo concessogli (ricordiamo che la gestione all'interno della singola coda di priorità è di tipo Round Robin, quindi è previsto un quanto di tempo) allora la sua priorità si alza, altrimenti si abbassa.

In sistemi multithreading vi sono altre facility interessanti che permettono di settare quella che viene chiama "affinità" per ogni singolo thread. Tramite la syscall SetThreadAffinity si può infatti fornire una maschera di bit per indicare al thread su quali CPU può entrare e quali invece gli sono precluse, controllo capillare necessario quando si sta parlando di sistemi multiprocessore.

Per quanto concerne il problema, ipotizziamo che il processo Pi sia generato all'istante Ti-esimo, dove T3 dista esattamente h ms da T0 e k ms da T2. Ergo, al fine di liberare una partizione ampia esattamente 2MB (in modo da ridurre al minimo la frammentazione esterna) la condizione è che P1 termini in un intervallo di tempo minore di h, e P2 in un intervallo di tempo minore di k. Non si contempla l'ipotesi in cui è P4 ad uscire dalla memoria di lavoro in quanto:

1) Non è richiesto;
2) Aumenterebbe la frammentazione esterna, in quanto rimane una partizione libera di 2MB inutilizzata.

Domanda 2 (5.25 punti)

Descrivere cosa sia un buffer cache e quali obiettivi siano legati al suo utilizzo. Descrivere inoltre il funzionamento di un buffer cache a sezioni multiple.

Si tratta di un'area di memoria trattenuta con lo scopo di ridurre (applicando in maniera intelligente politiche di rimpiazzo delle informazioni) il numero di accessi alla memoria. Il meccanismo essenziale su cui si basa questa tecnologia è il *principio di località* (che può a sua volta essere locale o temporale, ma non ci interessa in questa sede di trattazione), il quale afferma che, se un'applicazione ha recentemente avuto bisogno di un dato, molto probabilmente necessiterà anche di quelli contigui in un ridotto intervallo di tempo. Esempio: sto scorrendo un array con un ciclo while. Avendo fatto accesso alla entry i-esima, è molto, molto probabile che io voglia accedere alla entry i+1-esima.

Il buffer cache mantiene quindi informazioni per il momento rilevanti all'applicazione in esecuzione, aggiornandoli continuamente avvalendosi di due algoritmi principali:

1) Least Recently Used: ad ogni associazione è associato un timer, e, quando il buffer cache è pieno e l'applicazione chiede nuovi dati (ovvero, dati che non sono attualmente nel buffer cache e non è possibile depositarvi nulla poiché il buffer cache è pieno) quell'informazione con il timer più lungo (il timer viene resettato ogni volta che l'informazione viene nuovamente impiegata dall'applicazione) viene scelta come vittima;
2) Least Frequently Used: a livello stack si mantiene un counter per ogni informazione, che specifica quante volte questa sia stata effettivamente utilizzata. La vittima sarà quell'informazione con il counter più basso.

La vittima viene mandata in swap out dal buffer cache. Per farlo non ci si avvale del Virtual File System. I motivi sono molteplici, figurano tra di esse una gestione potenzialmente inefficiente dovuta in parte alle dimensioni non predefinite di quest'area. Ci si avvale, quindi, di quella che viene chiamata un'area di swap, la quale prevede una dimensione predefinita e la possibilità di elevata frammentazione interna.

L'anatomia del buffer cache inizialmente prevedeva due sezioni, una vecchia ed una nuova. Quando un'informazione si trova nella zona vecchia e viene utilizzata, il suo counter viene incrementato e viene trasportata nella sezione nuova. Qui il counter non viene più incrementato. Al momento di scegliere una vittima, viene scelta nella zona vecchia quell'informazione con il counter più basso e viene buttata fuori dal buffer cache; al contempo, l'informazione con il buffer più basso in sezione nuova passa in sezione vecchia, per poter poi essere eventualmente ricaricato in sezione nuova quando dovesse venire utilizzato nuovamente.

La successiva evoluzione del buffer cache implica l'utilizzo di una sezione intermedia. Quando un'informazione nella zona vecchia viene referenziata, il suo counter viene incrementato e viene passata direttamente in sezione nuova. Quando invece viene scelta una vittima nella sezione nuova, essa non passa direttamente in sezione vecchia, bensì passa nella sezione intermedia. Sia in sezione vecchia che intermedia i counter vengono incrementati.

2019-09-03:

Domanda 4 (5.25 punti)
Si consideri un insieme di 4 processi {P1, P2, P3, P4} ciascuno dei quali legge periodicamente un nuovo messaggio da uno slot di una memoria condivisa M. Il processo Pi legge esclusivamente dal corrispettivo slot M[i] della memoria condivisa. Un ulteriore processo PROD scrive periodicamente in ordine circolare i messaggi che devono essere letti dai processi Pi. Ogni 2 nuovi messaggi scritti, PROD attende che i processi destinatari li abbiano letti prima di poter procedere a produrre e scrivere altri messaggi sulla memoria condivisa M. La lettura da parte dei processi Pi dagli slot della memoria condivisa M è bloccante in assenza di nuovi messaggi, così come l'attesa di PROD dell'avvenuta lettura. Si schematizzi la soluzione del suddetto problema di sincronizzazione, usando solo semafori, fornendo lo pseudo-codice delle procedure LEGGI usata dai processi Pi e SCRIVI usata dal processo PROD.

Ho bisogno dei seguenti semafori:

1) Writer: {4 entry, 1 token l'una inizialmente tutte impostate a 1} -> Ho bisogno di 4 entry perché devo assicurare un ordine di scrittura circolare; inoltre, se le entry non fossero inizialmente impostate ad 1, PROD non avrebbe modo di scrivere perché la WAIT non avrebbe mai successo;
2) Go: {1 entry, 2 token inizialmente validi per PROD} -> Il semaforo precedente serviva per sincronizzare l'accesso concorrente alla risorsa. Leggasi: PROD può scrivere sul frammento i-esimo di memoria solo quando Pi lo autorizza, ergo, quando ha finito di leggere. Tuttavia, la seconda condizione è che PROD scriva solamente quando i due processi cui ha appena consegnato le informazioni hanno finito di leggere, ergo ho bisogno di questo semaforo. Si sarebbe potuto anche fare che i due processi appena interpellati fanno una SIGNAL sulla entry i+1-esima del semaforo Writer, ma questo può potenzialmente permettere a PROD di sincronizzarsi quando uno solo dei due ha letto e non tutti e due;
3) Readers: {4 entry, 1 token l'una inizialmente tutte impostate a 0} -> Ho bisogno di garantire che ogni processo Pi legga solo una volta (i.e: se facessi un semaforo con una entry e 4 token c'è il rischio che un processo prelevi due o più token, provocando starvation su altri processi in primis, e poi andrebbe a leggere un'area in cui potenzialmente non è ancora stato depositato nulla), ed all'inizio ogni frammento di memoria è vuoto, quindi è necessario che i processi Pi falliscano la WAIT finché non è PROD a fare una SIGNAL:

SCRIVI:

```
i= 0;
// Entro nella sezione critica di codice:
while(1){
        WAIT(Go, 2);
        // I processi precedenti hanno letto le informazioni;
        WAIT(Writer, entry i-esima);
        // Il processo su cui attualmente devo scrivere ha letto le informazioni che precedentemente avevo depositato, posso appropriarmi della risora;
        <Deposita il messaggio sulla entry i-esima della memoria condivisa>;
        SIGNAL(Readers, entry i-esima);
        i= (i == 4 ? 0 : i+1); // Garantisco la scrittura circolare;
}
```

LEGGI:

```
WAIT(Readers, entry i-esima);
<Leggi l'informazione dal frammento di memoria i-esimo>;
SIGNAL(Writer, entry i-esima);
SIGNAL(Go, 1);
```

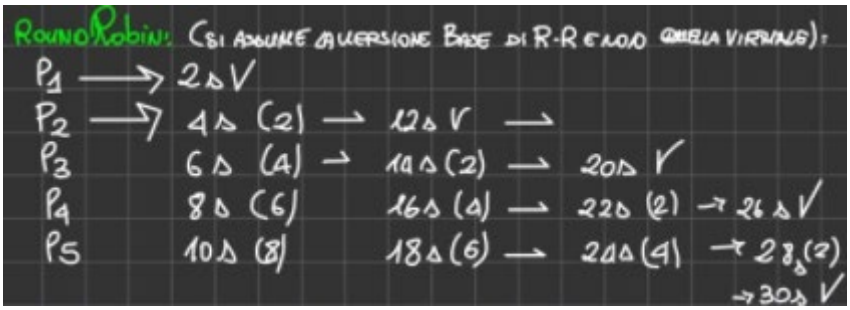
Domanda 1 (5.25 punti)
Si descrivano gli algoritmi di scheduling della CPU Fist-Come-First-Served (FCFS), Shortest-Process-Next (SPN), e Round-Robin (RR), discutendone in modo comparativo vantaggi e svantaggi. Si consideri inoltre uno scenario con 5 processi {P1,..., P5}, tutti di natura puramente CPU-bound, che vengono generati in sequenza a partire da P1 (ma senza ritardi tra l'uno e l'altro) le cui richieste di uso della CPU per completare la loro esecuzione sono le seguenti:

1) P1 richiede 2 sec.
2) Pi richiede 2 sec. in più rispetto del tempo richiesto da P(i-1)

Si calcoli, per tutti e tre gli algoritmi FCFS, SPN e RR il ritardo per il primo accesso alla CPU e il tempo di completamento del processo P5, supponendo che RR abbia un time-slice pari a 2 sec. e supponendo anche che il ritardo di context-switch tra un processo e un altro sia trascurabile.

Nota: nella trattazione a seguire, $I|O\ bound = IO, CPU\ bound = CPU$.
FCFS: è l'algoritmo di base, prevede che i processi vengano computati nell'ordine esatto in cui si trovano nella coda di attesa. Presenta notevoli svantaggi, quali mancanza di prelazione, starvation di processi IO (se un processo IO si trova di fronte in coda d'attesa diversi processi CPU, ipotizzando un tempo di esecuzione costante per i processi CPU e n processi CPU davanti al processo IO, dovrà attendere un tempo pari ad n*CPU_burst, ovvero un tempo lunghissimo). La successiva evoluzione è il Round Robin, che implementa una forma primitiva di prelazione (sebbene il Round Robin venga, ancora ad oggi, impiegato nello Scheduler Windows nella gestione della singola coda di priorità), ovvero ad ogni processo viene assegnata una quantità di tempo detta "time slice", allo scadere della quale il processo è costretto ad abbandonare la CPU anche se non ha terminato la sua esecuzione. Questo provoca starvation di processi IO, poiché lo slice deve essere abbastanza ampio da permettere ad un processo CPU di massimizzare la sua efficienza in CPU. Ergo, molto probabilmente un processo CPU usufruirà del suo slice al completo per molte volte, mentre, vista la celerità dei processi IO, essi potrebbero non utilizzare tutto il quanto di tempo concesso. Una soluzione parziale, che comunque introduce notevoli problemi per quanto concerne la progettazione hardware, viene introdotta dal Round Robin virtuale, che introduce anche il concetto di due code di priorità (prelazionati, back from IO). Infine, STN è probabilmente il più inefficiente e difficoltoso da realizzare. Si deve infatti stimare il tempo che il processo attuale impiegherà in CPU, tramite un filtro del primo ordine di questo tipo: $S_{n+1} = \alpha ST_n + (1 - \alpha)S_{n-1}$, ovvero, allo stato più recente del sistema viene assegnato un peso maggiore, mentre allo stato remoto un tempo minore (è meno influente). Tuttavia, questo implica una complicazione a livello progettuale, e vi è necessariamente starvation di processi CPU, in quanto essi sono ovviamente in fondo alla coda di attesa, e se arrivano (come è plausibile, in applicazioni multimediali/interattive) altri processi IO, difficilmente andranno in esecuzione. La sua successiva evoluzione, Shortest Remaining Time Next introduce la possibilità di cofrontare se il CPU burst rimanente all'attuale processo in CPU (x) sia minore a quello di un nuovo processo in arrivo (y). Se x > y, allora il nuovo processo prende il posto di quello vecchio in CPU, altrimenti no. E' un modo di introdurre la prelazione.

Per quanto concerne il problema, abbiamo:
FCFS: 2+4+6+8+10= 30;
SRTN: identico, perché i processi vengono generati in ordine di durata;
Si allega una foto per il Round Robin:



Domanda 2 (5.25 punti)
Descrivere la tecnica dell'I/O bufferizzato, indicando quali siano i classici impiegni di questa all'interno del software sia del kernel di un sistema operativo che applicativo.
Essenzialmente è il funzionamento del Buffer Cache, si veda *domanda 2, 2019-07-24*.

Domanda 3 (5.25 punti)
Descrivere la tecnica del "write-back" in sistemi di gestione della memoria virtuale basati su paginazione, indicando anche quali siano i supporti hardware e software necessari per implementare tale tecnica.

DA FARE

Domanda 4 (5.25 punti)
Si considerino due insieme di processi A = {P1, P2, P3, ... Pn} e B = {Pn+1, Pn+2, ... Pm}. Ciascuno dei processi nell'insieme A scrive periodicamente un nuovo messaggio su uno slot di una memoria condivisa M. Il generico processo Pi appartenente all'insieme A scrive esclusivamente sul corrispettivo slot M[i] della memoria condivisa. I processi nell'insieme B leggono periodicamente i nuovi messaggi prodotti dai processi nell'insieme A. Ogni processo Pj appartenente all'insieme B deve leggere i nuovi messaggi prodotti da tutti i processi dell'insieme A solo se essi non siano già stati letti da alcun processo nell'insieme B. Altrimenti Pj deve rimanere in attesa che ogni processo appartenente all'insieme A scriva un nuovo messaggio. Una volta letti tutti i messaggi, Pj deve scrivere una risposta su un altro segmento di memoria condivisa R, la quale deve essere letta da tutti i processi appartenenti all'insieme A esattamente una volta. Prima di poter scrivere un nuovo messaggio, i processi nell'insieme A dovranno rimanere in attesa fino a che la risposta al loro ultimo messaggio non sia stata scritta da Pj su R. Si schematizzi la soluzione del suddetto problema di sincronizzazione, usando solo semafori, fornendo lo pseudo-codice delle procedure SCRIVI-LEGGI usata dai processi appartenenti all'insieme A e LEGGI-SCRIVI usata dai processi appartenenti all'insieme B.

Ho bisogno dei seguenti semafori:

1) Writers-A: {N entry, 1 token l'una inizialmente impostati ad 1} -> devo garantire che il mio messaggio venga letto prima di scrivere nuovamente, impossibile senza un array di semafori binari;
2) Readers-B: {N entry, 1 token l'una inizialmente impostati a 0} -> Servono solo per capire se su quella entry della memoria condivisa vi sono informazioni; ad ogni entry accederà un solo Processo dei B;
3) Reader-B: {1 entry, 1 token inizialmente impostato a 0}-> Lo uso per garantire che un solo processo B possa leggere;
4) Writer-B{1 entry, N token} -> I processi A fanno mano a mano SIGNAL su questo semaforo per permettere la WAIT del processo Pj B che ha finora letto;
5) Reader-A{N entry, 1 token l'una inizialmente impostato a 0} -> Per segnalare il processo A i-esimo che c'è la risposta. Avrei potuto utilizzare un semaforo con una sola entry ed N token, ma così non avrei avuto la certezza che tutti i processi A leggessero esattamente una sola volta.

SCRIVI-LEGGI (A):

```
WAIT(Writers-A, entry i-esima);
<Scrivi sul frammento i-esimo di memoria condivisa>;
SIGNAL(Readers-B, entry i-esima);
// Mettiti in attesa della risposta:
WAIT(Readers-A, entry i-esima);
<Leggi la risposta dalla memoria condivisa R>;
SIGNAL(Writer-B, 1);
```

SCRIVI-LEGGI (A):

```
WAIT(Reader-B);
// Mi sono aggiudicato il diritto di leggere (sono l'unico processo tra quelli di tipo B che può ora leggere);

i= 0;
for(i from 0 to N (N non compreso)){
        WAIT(Readers-B, entry i-esima);
        <Leggi informazioni dal frammento i-esimo di memoria condivisa>;
        SIGNAL(Writers-A, entry i-esima);
}

// Preparati a scrivere una risposta su R:
WAIT(Writer-B, N tokens da prelevare);
<Scrivi sulla memoria R>
SIGNAL(Readers-A, 1, tutte le entry);
```

Domanda 3 (5.25 punti)
Descrivere l'algoritmo dell'orologio per la selezione della vittima in sistemi di memoria virtuale basati su paginazione. Spiegare anche se tale algoritmo soffre (o non soffre) dell'anomalia di Belady.

Proprio come un orologio, prevede due "lancette". La prima è chiamata *selezione* e la seconda *reset*. L'algoritmo si avvale del bit di presenza, associato alla pagina che viene volta per volta considerata. La lancetta di reset scorre periodicamente l'insieme della

pagine, impostando questo bit a 0, mentre la lancetta di selezione scorre l'insieme quando dovesse rivelarsi necessario trovare una vittima. In tal caso, sceglie la prima vittima con bit di presenza pari a zero. Ogni volta che una pagina viene interpellata, il suo bit di presenza viene posto ad 1. Il caso peggiore è ovviamente quello in cui tutte le entry sono impostate ad 1, e la lancetta della selezione fa un giro completo senza identificare un possibile candidato. In tal caso, la lancetta di Reset pone tutti i bit di presenza a zero, e quindi al prossimo giro la lancetta di selezione troverà la prima pagina come vittima. In tal caso l'algoritmo si riduce ad un semplice algoritmo FIFO; in fondo, l'algoritmo dell'orologio è una forma evoluta degli algoritmi FIFO (che soffrono dell'Anomalia di Belady, ovvero, all'aumentare dei frames aumentano i page fault), grazie al meccanismo di set e reset del bit di presenza (che viene assunto un po' come bit di età della pagina. In ogni caso, l'Algoritmo dell'Orologio rimane comunque un algoritmo di tipo FIFO, sebbene più efficiente, ed è quindi vittima dell'Anomalia di Belady: solamente gli algoritmi a Stack ne sono esenti).

Una versione più prestante dell'algoritmo (utilizzata dalla Apple) affianca al bit di presenza anche un *dirty bit*, il quale evidenzia se la pagina deve o meno essere copiata su disco una volta eliminata dalla memoria di lavoro. Ovviamente, la combinazione ideale di bit di presenza e dirty è 00, ovvero "puoi eliminare la pagina da memoria di lavoro e NON devi scriverla su disco, ergo NON devi fare accessi costosi alla memoria per questa pagina", che è quindi la migliore candidata.

Domanda 2 (5.25 punti)

Descrivere cosa sono gli "hard link" ed i "soft link" in un file-system. Inoltre, per il caso di file system UNIX, spiegare quali sono i supporti (le system-call) per creare/rimuovere hard/soft link e come questi link sono effettivamente implementati.

Hard Links: una generica entry di una tabella assomiglia a questo:

Nome del file	Record di Sistema (I-node in UNIX)

Questa entry non è altro che un hard link, ovvero un collegamento "forte" all'I-node associato ad un file. Essenzialmente, è il modo di *linkare* nome e I-node di un file. Da qui si evince che un file può assumere molti nomi, e che se cancello un hard link associato all'I-node di un file, questo non implica la cancellazione del file stesso, poiché ciò succede solamente quando:

1) Il numero di hard-links verso l'I-node scende a zero;
2) Il numero di collegamenti alla sessione verso il file scende a zero;

Potenzialmente, quindi, posso accedere ad un file anche se esso non figura più con un nome nel Virtual File System perché ho ancora un canale aperto verso la sessione.

Soft Links: nella sostanza è un nuovo file che contiene un pathname (quindi effettivamente un link) verso il file "originario". Al contrario degli hard-links, se il file originario dovesse essere cancellato, il soft link non avrebbe più senso di porsi, in quanto effettivamente starebbe linkando ad una risorsa ormai più disponibile. Per fare un esempio, è come avere una cartina con le indicazioni stradali verso un monumento, ma il monumento è stato distrutto.

Domanda 1 (5.25 punti)

Descrivere le principali caratteristiche di un sistema batch e di un sistema batch multiprogrammato, discutendo anche in modo comparativo i vantaggi o svantaggi dell'uno verso l'altro.

Sistemi Batch: si avvalgono di un set di moduli software detto *monitor*, presente in memoria di lavoro. Quando un programma viene fornito tramite un dispositivo di input, i suoi moduli vengono caricati dal monitor come delle subrutine del programma stesso, poi il monitor carica il programma in memoria di lavoro e lo avvia (ad esempio, agli albori dell'informatica, il lettore di schede perforate si interfacciava con il software del monitor); il controllo viene restituito al monitor nel momento in cui vi sia un errore, oppure il programma abbia terminato il suo task. Il problema di questi sistemi è *il bassissimo throughput*. Infatti, il monitor può caricare solamente un job (programma) alla volta in memoria di lavoro. Questo rappresenta un problema quando il job ha bisogno di intraprendere un'azione di I|O. In tal caso, infatti, vi sarà un tempo in cui il dato dovrà raggiungere il programma / il dato fornito dal programma dovrà essere consegnato, in cui essenzialmente la CPU non è attiva.

Sistemi Batch multiprogrammati: si inizia ad andare verso la multiprogrammazione (coesistenza contemporanea di più job su una macchina) con l'introduzione dello Spooling: questa tecnica implementa una nuova memoria, detta memoria disco, sulla quale l'input relativo al job attualmente in CPU viene anticipata e il suo output viene ritardato, garantendo un minore tempo di attesa medio (quindi un maggiore utilizzo percentuale della CPU) e coesistenza di dati di I|O appartenenti a diversi job. In ogni caso, solamente un job per volta è in grado di impiegare la CPU, tuttavia job diversi possono impiegare contemporaneamente dispositivi di I|O grazie all'introduzione della memoria disco. I vari job vengono memorizzati in memoria di lavoro econdo uno schema di partizioni multiple, nonostante questo porti a notevoli problemi di frammentazione esterna. Nonostante la multiprogrammazione introdotta dai sistemi Batch multiprogrammati riduca il tempo di risposta medio, aumenti il throughput e l'utilizzo percentuale dei dispositivi di I|O, il loro tallone d'Achille rimane il fatto che il passaggio del controllo viene fatto al monitor solamente in caso di terminazione del job, di errore o di richiesta di I|O, ergo quei job con molte richieste di I|O potrebbero essere penalizzati.

Anche se non richiesto, viene aggiunta anche la descrizione dei sistemi time-sharing e real time, che sono comunque argomenti papabili per una possibile traccia d'esame:

Sistemi time-sharing: essenzialmente, da questo momento in poi (anni 60'/70') si è iniziata ad approcciare l'idea di "scheduling" e di "prelazione". Il software del monitor potrebbe ad un certo punto ritenere necessario (in base alle politiche adottate, la più classica delle quali è la round robin) che è tempo di de-schedulare il job attuale dalla CPU per dare la possibilità ad un altro di andare in esecuzione. Questo è possibile grazie al supporto delle interrupts, che permettono la restituzione del controllo al software del monitor: mentre il job attuale sta eseguendo, casca in una interrupt, che trasferisce il controllo ad una routine di interrupt (avviene tramite interrupt vector, ovvero una struttura in cui sono memorizzati gli indirizzi delle routine del codice del monitor), ed ogni Sistema Operativo moderno ancora fornisce tale servizio. E' importante evidenziare che un sistema siffatto deve anche garantire la sicurezza che il job, una volta reschedulato in CPU, ricomincerà ad eseguire dall'istruzione successiva a quella di interrupt, ovvero bisogna salvare in un dato registro il valore PC+4 per questo job.

Sistemi real-time: viene introdotto il concetto di deadline, ovvero un limite massimo di tempo entro il quale il Sistema Operativo deve assicurarsi che un programma esegua specifici task. Si dividono in hard real time (ovvero, gravi conseguenze se la deadline non venisse rispettata. È utile per il controllo di processi industriali, ma non è adatto per processi interattivi), e soft real time (ovvero, la deadline dovrebbe essere rispettata, ma non vi sono conseguenze eccessive in caso non vi si riuscisse. In ogni caso, questi sono perfetti per applicazioni multimediali, e vengono anche impiegati da sistemi time-sharing).

2020-01-20:

Domanda 4 (5.25 punti)

Si consideri un insieme di N processi {P0, P1, P2, P3, …., PN-1}, ed una memoria condivisa M composta da N slot. Il processo P0 periodicamente attende che un nuovo messaggio venga depositato in M[0] e quando questo avviene, P0 legge il messaggio e lo riporta ad uno qualsiasi degli altri processi Pi scelto a caso, scrivendolo in M[i]. Sucessivamente P0 attende che il messaggio sia stato letto da Pi. Un ulteriore processo PROD periodicamente produce il messaggio destinato a P0, da depositare in M[0], e successivamente attende che P0 ed il processo Pi scelto da P0 abbiano entrambi letto tale messaggio. La lettura della memoria condivisa M è bloccante in assenza di nuovi messaggi. Si schematizzi la soluzione del suddetto problema di sincronizzazione, usando solo semafori, fornendo lo pseudo-codice delle procedure SCRIVI e LEGGI usate, rispettivamente, da PROD e da ciascuno dei processi Pi.

Ho bisogno dei seguenti semafori:

1) Writer: {1 entry, 2 token inizialmente disponibili} -> Non devo garantire alcun particolare ordine di scrittura, mi basta che uno dei processi P1, …, Pn-1 si aggiudichi ambo i token e scriva; il motivo per cui sono necessari 2 token invece che uno è che così posso fare una SIGNAL nel processo P0 e nel processo da lui scelto, segnalando allo stesso tempo che ambo i processi hanno finito di leggere e che un nuovo processo può scrivere;
2) Readers-all: {N entry, 1 token ciascuna inizialmente impostato a 0} -> Potevo anche fare un semaforo da una entry per P0 ed un semaforo da N-1 entry per gli altri processi, ma sarebbe stato ridondante. In ogni caso, dato che Pi sceglie un processo, devo essere capillare nella segnalazione;

SCRIVI:

```
WAIT(Writer, 2);
<Deposita il messaggio sul frammento 0-esimo della memoria condivisa>;
SIGNAL(Readers-all, entry 0-esima);
```

LEGGI:

```
WAIT(Readers-all);
<Leggi il messaggio depositato sul frammento i-esimo di memoria condivisa>;

// Se sono il processo 0-esimo, devo rigirare il messaggio ad un processo a caso:
if(id== 0){
        <Scegli un processo a caso e memorizzane l'indice in una variabile intera "i">;
        // Posso sicuramente scrivere, perché PROD si è assicurato che il processo a cui ho prima passato il messaggio abbia prima
        // finito di leggere, e non c'è quindi possibilità che io sovrapponga la vecchia informazione con la nuova.
        <Deposita il messaggio nel frammento i-esimo di memoria condivisa>;
        SIGNAL(Readers-all, entry i-esima);
}
SIGNAL(Writer, 1);
```

Domanda 1 (5.25 punti)
Si descrivano gli scheduler di CPU Shortest-Process-Next (SPN) e la sua variante Shortest–Remaining-Time-Next (SRTN), evidenziandone i vantaggi e gli svantaggi.

Si veda *domanda 1 2019-09-03* (la risposta qui richiesta si trova verso la fine della risposta a quella domanda).

Domanda 2 (5.25 punti)
Descrivere le caratteristiche salienti del virtual-file-system Unix. Si consideri inoltre uno senario dove un processo P apra un file F (attualmente non in uso da parte di alcun processo) e poi esegua 2 fork(), indicare il numero delle sessioni di I/O verso il file F a valle dell'esecusione delle 2 fork() da parte di P.

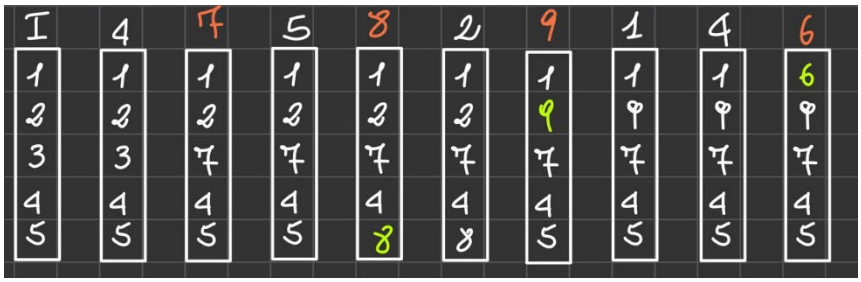
Caratteristiche più importanti di UFS (Unix File System) – nota: non vengono trattati tutti i dettagli relativi a canale di I|O, sessione e così via, segno solamente i tratti più importanti e distintivi):

1) Il File System tratta ogni file come uno stream di bytes (è quindi un modello di comunicazione stream I|O, ed è il motivo per il quale, quando uso una write o una read, posso specificare il numero di bytes che voglio vengano scritti/letti e controllare anche il valore di ritorno per comprendere se l'operazione è andata o meno a buon fine);
2) Il metodo di accesso al file è diretto (come su tutti i moderni Sistemi Operativi);
3) Il concetto di record di sistema è incarnato dall'I-node, il quale assolve agli stessi compiti del record di sistema, e il Sistema Operativo detiene un vettore di i-node;
4) Supporta il meccanismo delle extents Ext4, che permette di indicizzare blocchi contigui di un file (per un massimo di 128MB) con una sola struttura indice. Nota: questo non permette di risparmiare spazio per quanto concerne l'allocazione del file, bensì permette di ridurre il numero di metadati necessari nell'I-node al fine di indicizzare i blocchi del file;
5) Si utilizzano Access Control Lists (ACL) per specificare capillarmente i permessi di accesso ad un file. Essenzialmente, questa ACL altro non è che un file detto "*shadow*" associato al file originale (leggasi: c'è anche un I-node shadow che viene indicizzato dall'I-node originario); con i comandi di shell *getfacl* e *setfacl* si possono settare permessi di accesso ad un file per qualsiasi utente o gruppo di utenti.
6) La politica UNIX è "ogni dispositivo è un file", e come tali vengono visti ed utilizzati dal VFS;
7) Gli I-link relativi a file vengono copiati su disco rigido prima dello shutdown (in realtà la copia su disco rigido viene fatta spesso, perché se si dovesse perdere l'I-node, una volta chiusa la sessione attiva verso il file, non sarebbe più possibile trovarlo). Ciò non vale invece per gli I-nodes associati ai dispositivi.
8) Questa caratteristica del VFS UNIX ci permette di risolvere il problema proposto: quando un Processo UNIX si avvale di fork() per generare un processo figlio, il figlio eredita la tabella descrittori del padre (nota: ciò non è necessariamente vero in Windows. Infatti, in tal caso si può specificare se l'ereditarietà sia o meno ammessa nel programma) e i descrittori sono collegati alla stessa sessione, ergo abbiamo una sola sessione alla quale si collegano tre processi.

Domanda 3 (5.25 punti)
Descrivere l'algoritmo ottimo per la sostituzione delle pagine in ambiente di memoria virtuale. Si consideri inoltre una memoria di lavoro di 5 frame e la seguente sequenza di accessi a pagine logiche: 1 2 3 4 5 4 7 5 8 2 9 1 4 6; si determini il numero di page-fault nel caso di utilizzo dell'algoritmo ottimo. Si indichi infine se tale algoritmo soffra o meno dell'anomalia di Belady, motivando la risposta.

Sebbene irrealizzabile nella realtà (in quanto necessita che il Sistema Operativo conosca con esattezza i riferimenti *futuri* alle pagine, cosa ovviamente impossibile) viene utilizzato come termine di paragone quando vengono sviluppati nuovi algoritmi, in quanto è ovviamente l'algoritmo con efficienza massima. In sintesi, l'algoritmo seleziona per la sostituzione la pagina alla quale ci si riferirà dopo il più lungo tempo.

Per la soluzione del problema, si allega una foto:



In arancio si ha il numero di page miss (4) poi si hanno i primi page miss (I, ovvero i page miss sistemici dovuti al fatto che inizialmente la memoria di lavoro è vuota, per un totale di 8 page miss) mentre in giallo sono state segnate quelle sostituzioni "arbitrarie", ovvero, potevo anche sostituirle ad altre pagine tra quelle che non sono state in seguito menzionate nella sequenza data.

Infine, *suppongo* (attenzione, non ne ho la certezza assoluta!) che non soffra dell'Anomalia di Belady, in quanto non è un derivato delle FIFO. Infatti, una FIFO sceglie come vittima la pagina più "vecchia", qui invece ci proiettiamo nel futuro. Inoltre, a rigor di logica, conoscendo il futuro come faccio a rendere peggio di un FIFO? Infine, se viene scelto come metro di paragone per l'efficienza degli altri algoritmi ci sarà pure un motivo…

2020-02-24:

Domanda 4 (5.25 punti)

Si consideri un insieme di N processi {P0, P1, P2, P3, …., PN-1}, ed una memoria condivisa M composta da N slot. Ogni processo Pi legge esclusivamente dallo slot M[i] della memoria condivisa. Un ulteriore processo PROD produce messaggi per i pocessi {P0, P1, P2, P3, …., PN-1} e li scrive negli slot della memoria condivisa M. Ogni processo Pi è abilitato a leggere il suo messaggio solo dopo che tutti i messaggi destinati ai diversi processi in {P0, P1, P2, P3, …., PN-1} siano stati scritti da PROD. D'altro canto PROD può scrivere nuovi messaggi solo dopo che ogni processo in {P0, P1, P2, P3, …., PN-1} abbia letto l'ultimo messaggio scritto da PROD destinato ad esso.
Si schematizzi la soluzione del suddetto problema di sincronizzazione, usando solo semafori, fornendo lo pseudo-codice delle procedure SCRIVI e LEGGI usate, rispettivamente, da PROD e da ciascuno dei processi Pi.

Ho bisogno dei seguenti semafori:

1) Writer: {1 entry da N token, inizialmente tutti disponibili} -> Non mi avvalgo di N entry perché la scrittura può essere fatta solo dopo che tutti i consumatori hanno letto il messaggio, e l'ordine di scrittura non è particolarmente rilevante;
2) Readers: {N entry da N token l'una, inizialmente settato a zero} -> Ho bisogno di N token in quanto ogni processo può leggere solo dopo che gli altri hanno ricevuto il messaggio. Il processo PROD allora farà una segnalazione da 1 token a volta su ogni entry del semaforo lettore;

SCRIVI:

```
WAIT(Writer, N);
i= 0;
for(i from 0 to N-1){
   <Deposita il messaggio sul frammento 0-esimo della memoria condivisa>;
   SIGNAL(Readers, entry i-esima);
}
```

LEGGI:

```
WAIT(Readers, entry i-esima, N token da prelevare);
<Leggi il messaggio dal frammento i-esimo di memoria condivisa>;
SIGNAL(Writer, 1 token da depositare);
```

Domanda 1 (5.25 punti)

Si descrivano gli scheduler di CPU UNIX tradizionale e Windows, evidenziandone in modo comparativo i vantaggi e gli svantaggi.

Gli scheduler sono già stati descritti in *domanda 1 del 2019-07-24* (Windows) e *domanda 1 del 2019-06-25* (UNIX). In questa sede si discuteranno solamente vantaggi e svantaggi in maniera comparativa (si tenga conto che non c'è un simile confronto sulle slide del Professore, ergo questi sono i punti principali che mi son venuti in mente):

1) La gestione della priorità in UNIX è più "lenta"; infatti, sebbene vi siano degli impedimenti per rendere più competitivo un processo, alla fine dei conti essi possono variare tra tutte le fasce di priorità (da -19 a 20), mentre ciò in Windows non è possibile. Windows fornisce un controllo più capillare grazie al meccanismo di SetPriorityClass del processo generatore; uno svantaggio di Windows è la rigidità con cui vengono schedulati i processi della fascia real-time, i quali anche se si dimostrano molto leggeri (e ricordiamo, la politica di Windows decreta che se non esaurisci tutto il quanto di tempo la tua priorità aumenta) sono costretti a rimanere in una fascia più bassa. D'altro canto, thread molto pesanti (per la coda in cui si trovano) rimangono lì rallentando altri thread più veloci;
2) La gestione Round Robin che ambedue gli Scheduler impiegano nelle singole code si porta dietro gli storici problemi di Round Robin, ovvero la scelta del time slice. Windows in un certo modo pone rimedio (come abbiamo già detto, evolve la gestione della priorità di MLFQ, ovvero se un processo non termina il quanto di tempo la sua priorità viene alzata, altrimenti viene abbassata), sebbene esso non valga nel caso di *real time*. D'altro canto, lo scheduler UNIX SVR3 non propone alcun meccanismo per sistemare questo inconveniente;

Domanda 2 (5.25 punti)

Descrivere le caratteristiche salienti del file-system Windows.

1) Gli Hard-drive sono divisi in volumi detti partizioni e organizzati in cluster che vanno dai 512 ai 64K Bytes; tuttavia, per il memory management si usano generalmente 4K Bytes;
2) Ogni partizione è provvista di una MFT (Master File Table), ovvero l'equivalente Windows del vettore di I-node. La differenza sostanziale è che in UNIX era possibile montare più Virtual File System l'uno sull'altro creando una relazione gerarchica, mentre ciò in Windows non è possibile, ed è per questo che non è plausibile una sola MFT; altra differenza, è che mentre in UNIX ad ogni file corrispondeva un I-node, in Windows è possibile avere più entry nella MFT per un file (quindi, ogni file ha almeno un elemento nella MFT);
3) Cosa occupano le altre entry della MFT? Per spiegarlo, bisogna introdurre i file immediati, ovvero quei file i cui blocchi siano in memoria tutti contigui e che non occupino più di 4 entry della MFT. Una volta che questa dimensione è stata superata, si passa all'utilizzo di indici per indicizzare i cluster in cui è memorizzato il file. Questo è un vantaggio non indifferente di Windows: infatti, quando ho un file immediato e carico in cache i suoi metadati, ne carico anche tutto il contenuto, limitando l'accesso al disco rigido.
4) In genere Windows non viene utilizzato per i database a causa della "scrittura pigra", ovvero gli aggiornamenti vanno su hard-drive su base periodica, e questo ovviamente porta i progettisti a preferirgli UNIX;
5) La gestione del buffer cache segue uno schema Least Recently Used, e lo swap in/out dalla memoria è attuato da un thread di sistema.
6) Le ACL in Windows sono implementate mediante una lista di ACE (Access Control Entry). In un security descriptor sono implementate le entità "DACL – Discretionary ACL" (ovvero la specifica dei permessi mediante entità chiamate SID, di cui parleremo nel prossimo punto) e "SACL – System ACL" (con le quali si specificano le azioni di log da eseguire in base agli accessi).
7) SID: è una maschera di bits rappresentata come stringa; questo per poter essere eseguito su macchine diverse senza complicanze di progetto; inoltre, il SID si adatta a tutte le versioni di Windows. Essenzialmente, la sua struttura propone un campo iniziale per specificare la versione del SID (così si può riutilizzare lo stesso progetto nonostante le evoluzioni di Windows); il SID è poi relativo ad un utente o un gruppo, e può essere utilizzato solo in certi ambiti elencati nell'Autority Entity.

Domanda 3 (5.25 punti)

Descrivere l'algoritmo F-scan per la gestione delle interazioni con i dischi a rotazione. Si consideri inoltre uno scenario in cui arrivino al sistema operativo richieste per accedere alle seguenti tracce di un disco: 33 – 46 – 98 – 12 – 43 – 56 – 78 - 77 – 25. Si determini la sequenza effettiva di schedulazione delle operazioni verso il disco considerando che al più 4 richieste per volta possono essere immagazzinate nella coda di scheduling, e supponendo che la testina sia inizialmente posta sulla traccia 100 del disco.

Sebbene non richiesto, si parlerà di tutti gli algoritmi di scheduling del disco, in modo da fornire un quadro completo:
FCFS (first come first served): Richieste di I|O servite nell'ordine di arrivo, quindi non vi è preferenza di una richiesta rispetto ad un'altra (non vi è starvation), tuttavia non minimizza il tempo di attesa.

SSTF (shortest service time first): viene servita quella richiesta di I|O che implica il minor movimento della testina, tuttavia questo algoritmo è peggiore del precedente, perché (nonostante salvaguardi effettivamente la struttura meccanica del disco a rotazione) non minimizza il tempo di attesa (sebbene sia comunque minore di un FCFS) e può provocare starvation (ovvero, se arrivano in continuazione richieste di I|O sulla stessa traccia, ovviamente le altre non vengono servite. In generale, più si è lontani dalla traccia di partenza, meno è probabile essere serviti).

SCAN (algoritmo dell'ascensore: mentre gli algoritmi precedenti si potevano spostare avanti e indietro a seconda della policy adottata, l'algoritmo SCAN si sposta in una data direzione finché non arriva al termine delle tracce, o non sono finite le tracce in quella direzione, poi si gira. Questo può provocare starvation delle richieste generate nelle prossimità della traccia iniziale, e che tuttavia si sono palesate subito dopo il passaggio della testina. La variante CSCAN si sposta in una sola direzione, ricominciando daccapo laddove finiscano le tracce totali/richieste di I|O.

FSCAN (utilizzato da Linux): il problema degli algoritmi precedenti è che in situazioni patologiche (ovvero, i difetti prima descritti) la testina potrebbe essere bloccata per necessità in una zona ben definita, causando starvation di richieste relative ad altre tracce. FSCAN si avvale di due code distinte:

1) La schedulazione avviene sulla prima coda, dove vengono "bufferizzate" tutte le richieste di I|O;
2) Durante il passaggio di tali richieste alla seconda coda, in base all'algoritmo SSTN, le richieste vengono riordinate. Una volta che la sequenza di richieste nella seconda coda è ben determinata, nuove richieste vengono bufferizzate nella prima coda (ora libera) ma non vengono inserite nella seconda finché tutte le richieste attuali non vengono soddisfatte.

Per quanto concerne il problema (siano C1, C2 == coda 1,2);

Turno 1: testina in posizione 100:
C1= 33, 46, 98, 12;
C2= 98, 46, 33, 12;

Turno 2: testina in posizione 12:
C1= 43, 56, 78, 77;
C2= 43, 56, 77, 78;

Turno 3: testina in posizione 78:
C1= 25;
C2= 25;

2018-06-04:

Domanda 4 (5.25 punti)
Si considerino due gruppi di processi (A1, ... , An) e (B1,...,Bm), i quali utilizzano due segmenti di memoria condivisa M_A ed M_B per scambiare informazioni. Il generico processo Ai scrive periodicamente un nuovo messaggio sul segmento di memoria condivisa M_A mentre il generico processo Bj scrive periodicamente un nuovo messaggio nel segmento di memoria condivisa M_B. Quando i generici processi Ai e Bj hanno scritto il loro messaggio, ciacuno di essi attende e poi legge il messaggio scritto dal processo dell'altro gruppo. In particolare, Ai attende e legge il messaggio scritto da Bj, e Bj attende e legge il messaggio scritto da Ai. Entrambi poi scrivono una risposta nel segmento di memoria condivisa associato all'altro gruppo. In particolsare Ai scrive la sua risposta in M_B destinata a Bj, mentre Bj scrive la sua isposta in M_A destinata ad Ai. Quando la risposta è disponibile, ciascuno dei due processi la legge. Si schematizzi la soluzione del suddetto problema di sincronizzazione, usando solo semafori, fornendo lo pseudo-codice delle procedure SCRIVI-RISPONDI-GRUPPO-A e SCRIVI-RISPONDI-GRUPPO-B usate rispettivamente dai generici processi Ai e Bj.

Ho bisogno dei seguenti semafori:

1) Writer-A: {1 entry, 1 token inizialmente disponibile}; -> Essenzialmente (e questo vale anche per il prossimo semaforo) è una battle royale per aggiudicarsi la risorsa, quindi non mi interessa un particolare ordine in scrittura, o una qualche sorta di faireness; uno stesso processo potrebbe anche aggiudicarsi il token due volte di file;
2) Writer-B: {1 entry, 1 token inizialmente disponibile};
3) Reader-A: {1 entry, 0 token inizialmente disponibili, ma verrà poi settato ad 1 quanto l'informazione diviene disponibile};
4) Reader-B: {1 entry, 0 token inizialmente disponibili, ma verrà poi settato ad 1 quanto l'informazione diviene disponibile};

SCRIVI-RISPONDI-GRUPPO-A:

```
WAIT(Writer-A, 1); //
<Scrivi l'informazione nella memoria condivisa A>;
SIGNAL(Reader-B,1);

// Mi metto in attesa del messaggio sulla MB:
WAIT(Reader-A, 1);
<Leggi il contenuto della memoria MB>;
<Scrivi la risposta in MB>
SIGNAL(Reader-B,1);

// Mettiti in attesa di una risposta su Ma:
WAIT(Reader-A, 1);
<Leggi il contenuto dalla memoria Ma>;
SIGNAL(Writer-A, 1);
```

Il processo B è identico, cambiando ovviamente le dovute "A" con "B";

Domanda 2 (5.25 punti)
Si decriva il metodo di allocazione dei file indicizzato. Si supponga di avere un file system che supporta il metodo di allocazione indicizzato, in cui il record di sistema associato ad ogni file mantega 128 indici diretti, 4 indici indiretti e 4 indici doppiamente indiretti. Si supponga inoltre che il dispositivo di memoria di massa ove il file system è ospitato abbia blocchi di taglia pari a 1024 record, e che un indice di blocco di dispositivo sia espresso con 8 record. Si indichi la massima taglia possibile (in termini di numero di record) per un generico file allocato su dispositivo secondo tale schema di indicizzazione.

Nota: stiamo parlando di ALLOCAZIONE di file, non di ACCESSO. Ergo, ci riferiamo a BLOCCHI e non ai RECORDS.

In ogni caso, il metodo indicizzato è un'evoluzione dell'allocazione a catena. In questo caso, il Record di Sistema (l'I-Node nel caso di UNIX, la Master File Table nel caso di Windows) detiene dei puntatori a tutti i blocchi del file. Questo schema ha:

1) Il vantaggio di permettere allocazione non contigua senza dover scendere ai compromessi dettati dall'allocazione a catena (spazio sottratto nel blocco per trattenere un puntatore al prossimo blocco, per tornare ad un blocco precedente devo ripartire dall'inizio (blocco "head") e così via), mantenendo quindi la reale occupazione del file sebbene la disposizione non sia contigua. Nota: come già detto, la disposizione a catena prevedeva dei puntatori, e questo non permetteva il mantenimento della vera dimensione del file in memoria, poiché il blocco non era disponibile solo per l'informazione;
2) Lo svantaggio di appesantire di parecchio il record di sistema.

Una soluzione è lo schema indicizzato a più livelli, che permette di unire il buono di ambo le tecniche pagando un costo in svantaggi tutto sommato accettabile.

N.B probabilmente la traccia d'esame era scritta male, perché il problema poi cita "livelli indiretti", che sono propri dell'allocazione indicizzata a livelli multipli, non di quella indicizzata richiesta dalla traccia. In ogni caso, prima sono state descritte ambo le tecniche.

Soluzione del problema: FINIRE$Dmax = (128$

Domanda 3 (5.25 punti)
Descrivere la tecnica di gestione delle memoria basata su partizioni dinamiche, indicando anche di quali supporti per il binding degli indirizzi questa necessita.

Al contrario delle tecniche più "antiche" di partizionamento, le partizioni dinamiche possono essere utilizzate in numero e taglia variabile. Inizialmente la memoria di lavoro è vuota, e quindi il Sistema Operativo, idealmente, gestisce un "unicum", che sarebbe null'altro che una gigantesca partizione libera. Questa partizione viene periodicamente occupata da nuove partizioni, ovvero delle zone di memoria concesse ai processi, e la dimensione di queste zone è esattamente pari alla dimensione dell'Address Space del processo. Quando un processo ottiene una partizione, la base e il limite della stessa vengono mantenute in struct del sistema operativo (nota: queste dimensioni permettono di spiazzarsi per raggiungere lo spazio del processo desiderato, e di utilizzare il registro limite della MMU per non sforare in un'area di memoria logicamente invalida); quando libera la partizione, lo spazio adesso disponibile è visto come una nuova partizione libera. Il problema è che con il tempo si possono creare tanti frammenti liberi inutilizzabili a causa della dimensione dell'Address Space dei processi. Questo fenomeno, chiamato *frammentazione esterna*, viene "combattuto" dal Sistema Operativo mediante l'utilizzo di una costosa (ed infatti si sono studiati parecchi algoritmi di inserimento in memoria di lavoro dei processi, come First, Best, Worst Fit per limitare l'avvenire del fenomeno) tecnica nota come *ricompattazione*. Periodicamente (e la scelta tra gli algoritmi prima proposti -di cui il migliore è il worst fit, ovvero lo spazio più grande che può ospitare un processo tra quelli disponibili- è volta a far sì che questo periodo tra una ricompattazione e l'altra sia il più ampio possibile) il sistema operativo *trasla* le informazioni in modo da creare una mega-partizione libera.

Il supporto di binding necessario è ovviamente quello del Binding a tempo di Esecuzione, e quindi il supporto hardware della MMU, perché altrimenti non si potrebbe fare ricompattazione sennon ricaricando/ricompilando tutti i software e non si potrebbe far accedere un processo precedentemente estromesso dalla memoria in una nuova partizione.

Domanda 1 (5.25 punti)
Si descriva lo scheduler di CPU Windows.

Già risolto.

2018-06-27:

Domanda 4 (5.25 punti)
Si considerino due gruppi di processi (A1, ... , An) e (B1,...,Bm) i quali utilizzano un unico segmento di memoria condivisa M per scambiare informazioni. Il segmento di memoria condivisa M ha n distinti slot, ed il generico processo Ai scrive periodicamente nello slot M[i]. Quando tutti i processi Ai hanno scritto una nuova informazione nei relativi slot M[i], un generico processo Bj può leggere l'intero contenuto di M (quindi tutti gli slot). Il generico processo Bj non deve essere abilitato a leggere il contenuto di M se un processo del suo stesso gruppo (ad esempio Bk) ha già letto questo stesso contenuto in precedenza. In altre parole, l'intero cuntenuto di M deve essere visto da parte dei processi (B1,...,Bm) come un'unica informazione "consumabile" una sola volta. D'altra parte ogni generico processo Ai non può scrivere una nuova informazione in M[i] prima che quella che lui ha scritto in precedenza sullo stesso slot M[i] non sia stata letta (consumata) da un generico processo Bj. Si schematizzi la soluzione del suddetto problema di sincronizzazione, usando solo semafori, fornendo lo pseudo-codice delle procedure SCRIVI e LEGGI usate rispettivamente dai generici processi Ai e Bj.

Ho bisogno dei seguenti semafori:

1) Writers-A: {N entry, 1 token l'una inizialmente disponibile} -> Devo garantire ordine in scrittura; inoltre, una entry della memoria non può essere aggiornata finché non è avvenuta la lettura della stessa;
2) Reader-B: {1 entry, N tokens inizialmente disponibili}-> Mi serve solo per decretare quale tra i processi B si aggiudicherà il diritto di leggere la memoria;
3) Readers-B: {N entry, inizialmente 0 token l'una, ma sono aggiornati all'occorrenza inserendone 1 per entry} -> Segnala che c'è informazione nella entry i-esima della memoria;

SCRIVI:

```
WAIT(Writers-A, entry i-esima);
<Scrivi nel frammento di memoria condivisa i-esimo>;
SIGNAL(Readers-B, entry i-esima, 1);
```

LEGGI:

```
WAIT(Reader-B, N); // Se riesco a prelevare N token, significa che il precedente processo ha fatto N letture;
// Mi sono aggiudicato il diritto di leggere dalla memoria.
Int i= 0;
while(i from 0 to N (N non compreso)){
        WAIT(Readers-B, entry i-esima);
        <Leggi il contenuto del frammento i-esimo di memoria condivisa>;
        SIGNAL(Writers-A, entry i-esima); // Essenzialmente permetto a questo processo di scrivere anche se non lo leggerò io, ma quel processo che si aggiudicherà gli N token di Reader-B, così risparmiamo tempo;
        SIGNAL(Reader-B, 1 token);
}
```

Domanda 3 (5.25 punti)
Descrivere l'algoritmo dell'orologio per la selezione della "vittima" in sistemi di memoria virtuale basati su paginazione. Indicare inoltre se questo algoritmo soffre dell'anomalia di Belady.

Già risolto in *domanda 3 del 2019-09-16*.

Domanda 1 (5.25 punti)
Descrivere lo scheduler di CPU Unix tradizionale.

Già risolto, si veda *domanda 1 del 2019-06-25* (la risposta è più completa, ci sono anche le syscall per programmarlo);

Domanda 2 (5.25 punti)
Si decriva il metodo di allocazione dei file a catena. Si supponga di avere un file system che supporta il metodo di allocazione a catena. Si supponga inoltre che il dispositivo di memoria di massa ove il file system è ospitato abbia blocchi di taglia pari a 4 K record, e che un indice (puntatore) di blocco di dispositivo sia espresso con 16 record. Si supponga inoltre di avere un file F di taglia pari a 16 M record. Si calcoli il numero di blocchi necessari ad allocare il file sul dispositivo di memoria di massa secondo lo schema a catena.

L'allocazione a catena si comporta come una lista singolarmente concatenata, e ne porta quindi gli svantaggi. In primis, il Record di Sistema detiene l'informazione di qual è il "nodo testa" di questa catena. Da quel momento in poi, ci si può spostare al blocco successivo seguendo la catena di puntatori. Gli svantaggi sono i seguenti:

1) Il numero di blocchi allocati non rappresenta la vera dimensione del file. Infatti, una parte del blocco è occupata dal puntatore al blocco successivo, ergo non tutto lo spazio nel blocco è utilizzato dall'informazione effettiva;
2) Se mi trovo al millesimo blocco e devo tornare al blocco precedente, devo ritornare al nodo testa e ripartire da lì fino a raggiungerlo. Questo approccio è ovviamente disastroso per due motivi: il primo è un discorso di efficienza e di latenza che sono puramente user-oriented; il più pericoloso è a livello meccanico. Infatti, i dischi con testina magnetica a rotazione si usurano meccanicamente con gli spostamenti, e il difetto appena descritto è per loro un tallone d'Achille.

Per quanto concerne il problema, abbiamo che l'informazione totale (in termini di records) detenibile in un blocco di memoria è pari a $4 * 1024 - 16 = 4080\ records$, ergo in totale ho bisogno di $2^4 * 2^{20} = 2^{24}; \frac{2^{24}}{4080} = 4113\ blocchi$, di cui l'ultimo ovviamente non sarà pieno al 100%.

2018-07-17:

Domanda 4 (5.25 punti)
Si consideri un sistema di 3 processi concorrenti, A, B e C, i quali accedono ad un segmento di memoria condivisa M. Periodicamente i processi A e B aggiornano, rispettivamente, la prima metà e la seconda metà del segmento di memoria condivisa. Quando l'aggiornamento è stato eseguito, il processo A riscrive il contenuto dell'intero segmento M invertendo l'ordine dei byte. Il processo C legge periodicamente l'intero contenuto del segmento di memoria condivisa M; la lettura è ablitata solo dopo che l'aggiornamento da parte di A e B, e l'inversione da parte di A, siano stati effettuati. Nessun nuovo aggiornamento può avvenire su M da parte di A e B prima che C abbia letto il contenuto registrato in M. Si schematizzi la soluzione del suddetto problema di sincronizzazione, usando solo semafori, fornendo lo pseudo-codice delle procedure SCRIVI, usata dal processo B, SCRIVI –INVERTI, usata dal processo A, e LEGGI, usata dal processo C.

Ho bisogno dei seguenti semafori:

1) Writer-A: {1 entry, 1 token inizialmente disponibile} -> Comunica al processo A che può nuovamente scrivere;
2) Writer-B: {1 entry, 1 token inizialmente disponibile};
3) Inverter: {1 entry, inizialmente 0 token disponibili} -> Utilizzato da B per comunicare a A che ha finito di scrivere e, una volta che anche lui avrà terminato, potrà invertire l'informazione;
4) Reader: {1 entry, inizialmente 0 token disponibili} -> Il token verrà passato con una SIGNAL da A quando avrà terminato l'inversione dell'informazione;

SCRIVI:

WAIT(Writer-B, 1 token da prelevare);
<Scrivi sulla seconda metà della memoria condivisa M>;
SIGNAL(Inverter, 1 token da inserire);

SCRIVI-INVERTI:

WAIT(Writer-A, 1 token da prelevare);
<Scrivi sulla prima metà di memoria condivisa M>;
WAIT(Inverter, 1 token da prelevare);
<Inverti l'informazione>;
SIGNAL(Reader, 1 token da inserire);

LEGGI:

WAIT(Reader, 1 token da prelevare);
<Leggi tutta la memoria condivisa M>;
SIGNAL(Writer-B, 1 token da inserire>;
SIGNAL(Writer-A, 1 token da inserire>;

NOTA: Invece che fare due semafori per i Writer, potevo farne uno solo a due entry, ma non essendovi ipotesi che questi due processi possano comprendere di essere lo 0-esimo e il 1-esimo (per accedere alle corrette entry del semaforo) ho preferito non farlo. Non ho ritenuto sufficiente l'informazione che uno accede al primo segmento della memoria ed uno al secondo, perché magari hanno gli indirizzi a quei frammenti e non un indice. Insomma, consiglio di confrontarsi con il Professore su questo punto.

NOTA: tutte le altre domande dell'appello sono già state affrontate precedentemente, ergo non verranno nuovamente risolte.
(P.S se mi avanza tempo vi scrivo anche dove trovarle precisamente, tuttavia, da ora in poi non caricherò le foto degli esercizi già svolti);

2018-09-07:

Domanda 4 (5.25 punti)
Si consideri un insieme di N processi concorrenti $\{P_0,...,P_{N-1}\}$, ciascuno dei quali scrive periodicamnete un nuovo messaggio un uno specifico slot di un segmento di memoria condivisa M avente esattamente N slot. Il generico processo Pi scrive esclusivamente sull'i-esimo slot M[i]. Un ulteriore processo COORDINATOR attende che tutti i processi Pi abbiano scritto il proprio messagio in M e poi scambia il contenuto degli slot di M secondo la regola del buffer circolare (M[i] viene caricato in M[(i+1)%N]). Una volta effettuato lo scambio, il processo Pi è abilitato a poter leggere il contenuto del relativo slot M[i], contenente l'informazione aggiornata. Al generico processo Pi non è permesso di scrivere un nuovo messagio in M[i] fino a che il suo ultimo messaggio scritto non sia stato scambiato da COORDINATOR.
Si schematizzi la soluzione del suddetto problema di sincronizzazione, usando solo semafori, fornendo lo pseudo-codice delle procedure SCRIVI-LEGGI, usata dai processi Pi, e SCAMBIA, usata dal processo COORDINATOR.

Ho bisogno dei seguenti semafori:

1) Writers: {N entry, 1 token l'una inizialmente disponibile}-> Garantisco l'ordine di scrittura e che ogni processo scriva esattamente una volta finché l'informazione non è stata swappata e poi letta;
2) Readers: {N entry, inizialmente nessuna ha token disponibili} -> Avverto un processo che il messaggio è stato swappato e può leggerlo;
3) Swap: {1 entry, inizialmente zero token disponibili, ma verranno aggiornati mano a mano fino ad N};

SCRIVI-LEGGI:

```
WAIT(Writers, entry i-esima, 1);
<Deposita il messaggio sul frammento i-esimo della memoria condivisa>;
SIGNAL(Swap, 1);

// Aspetta di poter leggere il messaggio:
WAIT(Readers, entry i-esima, 1);
<Leggi il messaggio>;
SIGNAL(Writers, entry i-esima, 1);
```

SCAMBIA:

```
int i= 0;
for(i from 0 to N (N non compreso)){
        <Scambia il contenuto del frammento i-esimo con il contenuto del frammento ((i+1)%N)-esimo>;
        SIGNAL(Readers, entry i-esima, 1);
        SIGNAL(Readers, entry ((i+1)%N)-esima, 1);
}
```

Domanda 1 (5.25 punti)

Descrivere lo scheduler di CPU multi-level feedback-queue. Si consideri inoltre uno scenario in cui tale scheduler abbia 4 livelli di priorità, ed in cui il quanto di tempo assegnato ai processi al livello 0 (entry level) sia di 1 millicondo. Si supponga inoltre che all'istante T0 nascano 2 processi A e B, entrambi CPU-bound. Il processo B richiede 10 millisecondi di tempo di CPU per completare la sua esecuzione. Si identifichi la durata massima (in termini di tempo di CPU) del processo A affinche il processo B possa completare la sua esecuzione entro il tempo T0+17 nei due casi in cui il primo processo ad essere schedulato in CPU sia A oppure B. Si supponga che il tempo di CPU per i context switch e per l'esecuzione dello scheduler sia nullo.

Il MLFQ è l'algoritmo "padre" dei moderni algoritmi di Scheduling UNIX e Windows. In effetti, l'idea di code di priorità distinte con gestione Round Robin all'interno di ognuna di esse presente in ambo gli attuali scheduler dei due sistemi nasce proprio in questa occasione. In particolare, MLFQ prevede livelli di priorità da 0 a N-1, e la priorità dipende dalla coda in cui ci si trova.

Il tallone d'Achille dell'algoritmo (cui Windows pone rimedio, almeno parzialmente) è che risulta possibile scendere in priorità ma non è possibile aumentarla. Se infatti un processo dovesse esaurire tutto il quanto di tempo concessogli in CPU, si dimostrerebbe più pesante, e verrebbe "degradato" alla successiva coda di priorità, e così via, finché il quanto di tempo concessogli non verrà che occupato solo parzialmente.

Il problema è, ovviamente, la starvation di processi molto lenti, quali i CPU bound. Una soluzione parziale a questa inconvenienza è la cessione di un quanto di tempo pari a 2^i, dove i è l'indice della coda in cui si trova il processo. Ergo, più si scende di priorità, più il quanto di tempo si espande, ma questo non limita le problematiche dovute alla poca faireness, in quanto processi più veloci (in generale, I|O bound) avranno sempre la meglio su un processo CPU bound. Immaginiamo infatti un processo CPU bound che, al prossimo turno, prenderà (finalmente) il suo turno in CPU, solo per vederselo sottratto dall'arrivo di un processo in una coda superiore alla sua. Questo è uno degli inconvenienti di MLFQ, ma la sua struttura è ancora oggi riconoscibile nei moderni schedulers.

Per quanto concerne il problema: in primis ci calcoliamo di quanti CPU burst necessita B indipendentemente dal caso citato. Andando avanti nelle varie code per ogni turno avremo (a sinistra il quanto concesso, a destra il tempo rimanente per il completamento):

$$(2^0 = 1ms, 9ms) \rightarrow (2^1 = 2ms, 7ms) \rightarrow (2^2 = 4ms, 3\,ms\,) \rightarrow (2^3 = 8ms,\ esecuzione\ completata)$$

In totale B ci mette 10ms, quindi per A rimangono soltanto 7ms di esecuzione al massimo, ma questo dipende da quando viene schedulato.

A schedulato per primo:

In tal caso avremmo (// significa che sono passati alla prossima coda di priorità):

A(1) + B(1) -> (A, B //, 2ms) -> A(2), B(2) -> (A, B //, 6 ms) -> A(4), B(4)-> (A, B // 14ms) -> B non ce la fa a finire in tempo! Infatti, anche se A ci mettesse un solo ms (non completando quindi tutto il quanto di tempo), B avrebbe da computare altri 3ms, ma per la deadline ne rimangono solo due.

Il tempo d'esecuzione massimo di A è quindi 7ms, ergo, dobbiamo supporre che termini la sua esecuzione ad A(4);

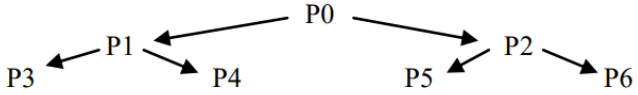
A schedulato per secondo:
B(1) + A(1) -> (A, B //, 2ms) -> B(2), A(2) -> (A, B //, 6 ms) -> B(4), A(4)-> (A, B // 14ms)-> B(8, ma prende solo i primi 3ms, quindi l'esecuzione è terminata)->A(3) -> In questo caso non vi sono limiti sulla durata di A.

N.B le altre due domande sono già state discusse nel resto del file.

2018-09-18:

Domanda 4 (5.25 punti)

Si consideri un insieme di processi {P0 .... P6} le cui relazioni sono organizzate come un albero binario secondo il seguente schema:



Ciascuno dei processi comunica con il "parent" nella gerarchia tamite un segmento di memoria condivisa. Indichiamo con Mi,j il segmento di memoria condivisa che permette al processo i-esimo (il parent) di comunicare con il processo j-esimo (ovvero il figlio). Il processo radice P0 produce periodicamente una nuova informazione che deve essere comunicata tramite memoria condivisa ai processi figli, e questi devono leggerla ed inoltrarla a loro volta verso i loro figli. Quando un processo tenta di leggere una nuova informazione, esso deve entrare in stato di attesa nel caso in cui tale informazione non sia ancora resa disponibile dal parent. D'altro canto, ogni processo Pi può inserire una nuova informazione in Mi,j solo dopo che l'ultima informazione inserita sia stata letta dal processo Pj. Si schematizzi la soluzione del suddetto problema di sincronizzazione, usando solo semafori, fornendo lo pseudo-codice delle procedure eseguite dai processi in {P0 ... P6} per implementare lo scambio di informazioni sopra descritto.

La difficoltà del problema sta nell'identificare il numero di entità in gioco. Supponiamo di poter determinare se il nodo ha o meno figli: in tal caso si identificano il nodo ROOT ed il nodo FIGLIO. A sua volta, il nodo FIGLIO dovrà controllare se possiede dei figli, e, in tal caso, passare loro le informazioni.

Una soluzione più semplice è individuare 3 entità: ROOT, MEDIANI e FIGLI. In questo modo è molto più semplice, ma vi è la necessità di portarsi appresso 3 funzioni. Tuttavia, è forse quella che più si addice alla richiesta di "usare solo semafori".

Per la seconda soluzione ho quindi bisogno dei seguenti semafori:

1) Write-Root: {1 entry, 2 token inizialmente disponibili} -> Potevo anche fare un semaforo a due entry con 1 token l'una. Questo effettivamente mi avrebbe permesso di capire chi ha letto e chi no, ma anche in questo modo, mettiamo caso P2 abbia terminato e P1 no, e io faccio prima la WAIT su P1, in tal caso P2 dovrebbe attendere comunque P1, quindi il vantaggio di poter trasferire una informazione ad un figlio indipendentemente da cosa abbia fatto l'altro figlio si va a perdere;
2) Readers-Mediani: {2 entry, token inizialmente non disponibili};
3) Readers-Figli: {4 entry, token inizialmente non disponibili};
4) Writers-Mediani{2 entry, 2 token ciascuna inizialmente disponibili};

In tal caso:

<u>SCRIVI-ROOT:</u>

WAIT(Write-Root, 2 token da prelevare);
<Scrivi il messaggio su M0,1>;
SIGNAL(Readers-Mediani, entry 0-esima);
<Scrivi il messaggio su M0,2>;
SIGNAL(Readers-Mediani, entry 1-esima);

<u>LEGGI-SCRIVI MEDIANI:</u>

WAIT(Readers-Mediani, entry (i-1)-esima);
<Preleva l'informazione da M0,i>;
WAIT(Writers-Mediani, entry (i-1)-esima, 2 token da prelevare);
<Fai il forwarding del messaggio ai tuoi figli>;
SIGNAL(Readers-Figli,

Domanda 3 (5.25 punti)
Descrivere i principali servizi di sistema operativo per l'allocazione di memoria virtuale su sistemi Unix e Windows.

UNIX: i servizi principali sono due, ovvero:

1) mmap(void *addr, size_t size, protection, flags) è la prima facility messa a disposizione da UNIX. Essa permette di mappare delle pagine direttamente sull'address space del processo chiamante. Non solo, permette anche di specificare come esse debbano essere mappate. Infatti, nel campo flags è possibile inserire i flags MAP_ANONYMUS, che essenzialmente mappa l'intera zona di memoria a zero (quindi una pagina vuota), in base ad un file (e quindi il contenuto del file è alla base della mappatura delle nuove pagine di memoria) oppure, e forse è la più importante, MAP_SHARED. In effetti, questo potente meccanismo permette di rendere "comune" una zona di address space tra processo padre e figlio. Infatti, i processi figli *copiano* l'address space paterno nel senso che quella memoria è di tipo *copy on write*. Questo significa che, alla prima operazione di scrittura, l'address space del processo figlio viene diviso da quello padre, in modo che le azioni del processo figlio non influenzino il processo padre. Tuttavia, MAP_SHARED permette di condividere un certo numero di pagine di memoria, e ogni modifica di padre e figlio sarà riportata su tali pagine.

2) Inoltre, UNIX mette a disposizione shmget; è una facility potente che permette di creare una zona di memoria condivisa tra più processi (ed anche threads, ovviamente) che utilizzino la facility shmat(che si avvale dell'identificatore logico ritornato da shmget) per "annettere" al proprio Address Space la memoria condivisa appena creata. I processi (e di conseguenza i threads generati da suddetti processi, dato che con essi condividono l'Address Space) collegati a tale memoria virtuale vedono quindi una zona comune con la quale scambiare dati. I processi hanno anche la possibilità di distaccare il proprio Address Space dalla shared memory grazie a shmdet. Infine, la facility shmctl con il flag SHM_RMID permette di rimuovere definitivamente la shared memory.

WINDOWS:

1) VirtualAlloc: molto simile a mmap, questa facility dei sistemi Windows permette di mappare delle pagine a zero (a meno che il flag MEM_RESET non sia specificato) nell'Address Space del processo chiamante. La cosa interessante è il controllo più capillare che questa facility offre grazie ai flags a seguire: MEM_COMMIT indica di mappare pagine e materializzarle; MEM_RESERVE indica di mappare pagine ("prenotarle") senza però materializzarle in memoria di lavoro. Il flag più interessante è senza ombra di dubbio MEM_RESET_UNDO, ovvero le pagine indicate attualmente non sono logicamente accessibili per scrittura e/o lettura, ed ora sono inutili. Tuttavia, a breve potrebbero tornare utili, ergo il processo piazza dei "placeholders" in modo da "salvaguardare" quelle pagine di memoria e poterle eventualmente reclamare. Con l'operazione inversa, VirtualFree è invece possibile 1) mandare in stato "riservato" le pagine indicate con MEM_DECOMMIT e liberare definitivamente le pagine con MEM_RELESE.
2) Windows poi offre OpenFileMapping per accedere ad un mapping di file già esistente, e MapViewOfFile per mappare delle pagine sull'address space del processo padre sulla base di un processo aperto.

Domanda 1 (5.25 punti)
Descrivere l'algoritmo di scheduling di CPU SPN (Shortest Process Next), discutendone vantaggi e svantaggi. Si consideri inoltre uno scenario in cui tale algoritmo venga impiegato per assegnare la CPU a 4 processi, A, B, C e D, tutti CPU-bound. Le durate (tempo di CPU richiesto) di questi sono rispettivamente 1, 2, 4 e 8 millisecondi. Si determini per ciascuno dei processi il tempo di attesa e il tempo di turnaround. Si determini inoltre il throughput garantito da tale algoritmo di scheduling nel suddetto scenario di carico, prendendo come riferimento la finestra temporale di esecuzione dei 4 processi.

L'algoritmo è già stato descritto in un'altra prova. Per quanto concerne il problema, definiamo:

1) *Tempo di attesa*: tempo che un processo attende nello stato ready, da non confondersi con il prossimo parametro;
2) *Tempo di turnaround* come quella quantità di tempo che intercorre dall'avvio di un processo (quindi dal momento in cui viene schedulato) all'inizio della produzione del suo *primo* output (un processo infatti ha -in genere- più output - soprattutto nella moderna informatica multithreading-);
3) *Throughput:* numero di processi completati per unità di tempo. In questo caso, la finestra temporale è quella di esecuzione dei 4 processi.

Ergo:
Tempo di attesa: (0, 1, 3, 7);
Tempo di turnaround: (1, 3, 7, 15);
Throughput: 4/15 è circa 1 processo ogni 2.66 millisecondi;

Alla domanda 2) è già stata data una risposta in un'altra prova d'esame.

Domanda 4 (5.25 punti)
Si consideri un sistema con un processo A ed un insieme di 4 processi {B1, B2, B3, B4}. Il processo A scrive periodicamente un nuovo messaggio su una memoria condivisa M. Dato un messaggio scritto da A, questo dovrà essere letto da esattamente 2 tra i processi Bi (senza un particolare ordine), mentre il successivo messaggio (in una coppia di 2 messaggi) dovrà essere letto dagli altri due processi Bi. Ogni processo Bi che tenti di leggere un messaggio deve entrare in blocco nel caso in cui il messaggio non sia disponibile, oppure nel caso in cui il messaggio correntemente scritto non possa essere letto da Bi poichè lo stesso Bi aveva letto il precedente messaggio scritto da A. D'altro canto, il processo A non potrà scrivere un nuovo messaggio sulla memoria condivisa M, dovendo così entrare in blocco, prima che l'ultimo messaggio da esso scritto non sia stato letto da almeno 2 processi Bi. Si schematizzi la soluzione a tale problema di sincronizzazione usando solo semafori, fornendo il pseudo-codice della della procedura SCRIVI utilizzata dal processo A e quello della procedura LEGGI usata dal generico processo Bi.

La difficoltà non è in realtà quella lapalissiana (ovvero permettere a due processi soli di leggere, perché per questo basta un semaforo a due token). Il problema è come impedire che, una volta caricati nuovamente 2 token nel semaforo dei lettori, gli stessi due processi iniziali (o uno di loro) si aggiudichino uno dei due token. Siccome si possono usare solo semafori, il meccanismo deve essere creato in modo che i primi lettori siano sottoposti ad una WAIT finché gli altri due non hanno finito. In questo modo, non possono ritornare a prendersi i token, perché sono in attesa su un semaforo. Questo permette anche di utilizzare la stessa funzione per ambedue i tipi di lettore, invece che fare due funzioni diverse.

1) Writer: {1 entry da 2 token inizialmente disponibili} -> I secondi processi lettori lo utilizzeranno per segnalare ad A di poter accedere nuovamente alla memoria per scrivere;
2) Readers: {1 entry con 0 token inizialmente disponibili, ma A ne caricherà due a volta} -> Serve per segnalare ai processi lettori che l'informazione è effettivamente presente;
3) Done: {1 entry con 0 token inizialmente disponibili, ma i processi lettori ne caricheranno uno a volta fino ad arrivare a 4} -> In questo modo avverto il processo A che ho finito di leggere. Quando il semaforo arriverà a 4 token totali, il processo A sarà in grado di fare una WAIT per sbloccare tutti i processi con il prossimo semaforo:
4) Go: {4 entry, inizialmente con tutti i token posti a zero, ma il processo A eventualmente li setta ad 1} -> In questo modo posso riutilizzare la stessa funzione LEGGI per tutti e due i processi (primi e secondi lettori) dato che i primi lettori vengono bloccati e non potranno assicurarsi i token che A inserisce in Readers dopo aver depositato il secondo messaggio.

SCRIVI:
```
WAIT(Writer, 2);
<Deposita l'informazione su M>;
SIGNAL(Readers, 2);
// Aspetta che i primi processi lettori ti diano il via:
WAIT(Writer, 2);
<Deposita un nuovo messaggio su M>;
SIGNAL(Readers, 2);
WAIT(Done, 4);
SIGNAL(Go, tutte le entry, 1 token);
```

LEGGI:
```
WAIT(Readers, 1);
// Mi sono assicurato un token tra i due disponibili insieme ad un solo altro processo;
<Leggi l'informazione dalla memoria condivisa>;
// Avverti il processo A che può tornare a scrivere:
SIGNAL(Writer, 1);
// Aggiungi un token al semaforo "done";
SIGNAL(Done, 1);
WAIT(Go, entry i-esima);
```

Domanda 2 (5.25 punti)
Descrivere il metodo sequenziale indicizzato per l'accesso ai file. Si consideri inoltre un file-system in cui i file sono allocati secondo uno schema contiguo, ed il dispositivo di memoria di massa abbia blocchi di capacità pari a 4096 record. Si determini la latenza massima di accesso ad un record di un file F sequenziale indicizzato che abbia taglia pari a 1M record, ed il cui file di indici f abbia taglia pari a 512 record, e contenga 128 chiavi. Per semplicità si supponga che il costo di gestione di ogni chiave, una volta caricata in memoria di lavoro, sia costante e pari ad 1 millisecondo e che il costo di caricamento di un blocco di dispositivo in memoria di lavoro sia anche esso costante e pari a 10 millisecondi.

Il metodo è già stato descritto in un'altra domanda. Per quanto concerne il problema:

Il file in memoria pesa esattamente 256 blocchi, allocati tutti in maniera contigua (ergo, il costo di ricerca di un blocco è lineare ed al massimo richiede di vagliare tutti i 256 blocchi possibili). Inoltre, il tempo totale per poter usufruire delle chiavi è pari a

$$C = tempo\ caricamento\ chiavi + tempo\ accesso\ a\ tutte\ le\ chiavi = 10 + 128 = 138ms;$$

Cerchiamo quindi il caso peggiore possibile, ovvero il tempo di accesso all'ultima area puntata.

Per quanto concerne la disposizione delle chiavi, abbiamo in realtà due casi, di cui:

1) Caso ottimo: ogni chiave punta a 256/128= 2 blocchi di memoria, ergo in totale 2*4096= 8192 records; essendo allocato contiguamente il costo d'accesso nel caso peggiore è $Tempo\ di\ caricamento\ dei\ blocchi\ nella\ memoria\ di\ lavoro + tempo\ di\ accesso\ all'ultimo\ record\ caricato + C = 2 * 10 + 2 * 4096 + L$;
2) Le prime 127 chiavi puntano ai primi 127 blocchi, mentre l'ultima chiave punta agli ultimi 129 blocchi. I blocchi vanno caricati tutti. 256*10 (tempo di caricamento di tutti i blocchi) ??????????????????????????

2020-06-16:

DOMANDA 1
Descrivere lo schema di gestione della memoria basato su paginazione. Inoltre, per il caso di memoria virtuale basata su paginazione a 2 livelli, in cui ciascuna tabella delle pagine a qualsiasi livello sia costituita da 512 elementi, si determini:
1) per indirizzi logici a 40 bit la struttura dell'indirizzo e l'utilizzo dei bit nel suddetto schema di paginazione;
2) il numero massimo delle pagine fisiche (frame) di memoria impegnate da un processo nel caso in cui il numero delle tabelle a qualsiasi livello correntemente usate per quel processo in suddetto schema di paginazione sia pari a 10.

Paginazione: la memoria di lavoro viene vista come un insieme di unità (dalla dimensione fissa) chiamate *frames*. In questi frame vengono caricate le cosiddette pagine, sulle quali vengono caricati i dati dei vari processi. Questo schema è vantaggioso in quanto permette di usufruire di spazi di indirizzamento non contigui: un processo potrebbe aver bisogno di 100 pagine e non per forza queste dovrebbero essere contigue, minando quindi il problema della frammentazione esterna. Rimane tuttavia il problema della frammentazione interna, ergo per una pagina di dimensione X bytes in genere la frammentazione interna assume un valore medio di X/2 Bytes (si può usare la paginazione segmentata per ridurre questo problema. Anche se non richiesto esplicitamente dalla domanda, ne parleremo in seguito per fornire un quadro completo).

Parliamo ora dei supporti alla paginazione. Ogni processo possiede una propria tabella delle pagine, sulla quale è indicato in quale frame si può trovare la pagina richiesta (perché, ricordiamo, non necessariamente tutte le pagine appartenenti ad un singolo processo sono contigue). Di fianco ad ogni pagina, sono presenti due bit: il bit di presenza (che indica se la pagina è effettivamente materializzata in memoria di lavoro) ed il dirty bit (questo indica se la pagina, una volta rimossa da memoria di lavoro, dovrà o meno essere copiata su disco). La presenza di questi due bit è fondamentale per gli algoritmi di sostituzione delle pagine. In ogni caso, si tenta di evitare, per quanto possibile, l'accesso alla tabella. Infatti, esiste un buffer di sistema noto come TLB nel quale vengono caricati gli indirizzi delle pagine utilizzate più di recente, in modo da dover evitare di accedere alla tabella. Ciò è voluto perché cercare una entry dalla tabella richiede l'impiego del firmware, ovvero microcodice specifico della CPU, rendendo quindi l'operazione potenzialmente costosa. L'accesso è comunque inevitabile in caso vi dovesse essere una miss sul TLB. In ogni caso, ogni volta che vi è un cambio di contesto, il TLB deve essere flushato. Infatti, se ciò non avvenisse, quando il processo P0 viene deschedulato e il processo P1 prende il suo posto in CPU, se il processo P1 chiede la pagina 0 e questa è presente nel TLB, in realtà P1 accederebbe alla pagina 0 del processo precedente.

Una possibilità interessante è quella di *bloccare* certi frame sulla memoria di lavoro, in maniera tale che la relativa pagina non possa essere swappata fuori dalla memoria (il kernel del Sistema Operativo è effettivamente istanziato su pagine bloccate).

In UNIX è presente inoltre una syscall nota come brk, che permette di spostare un elemento noto come il program break, ovvero quell'indirizzo che determina la fine della sezione .bss(dati non inizializzati). E' possibile spostare il program break, a patto tuttavia di dover (de)mappare pagina del processo in questione.

PARTE NON RICHIESTA: il problema della frammentazione interna della paginazione (abbastanza rilevante dato che corrisponde a circa metà dello spazio totale di una pagina) può essere risolto pensando allo spazio di indirizzamento (e quindi NON la memoria di lavoro, bensì lo spazio di indirizzamento di un processo, che ora è visto come un insieme di pagine) come un insieme di tanti piccoli segmenti di memoria (ergo, un insieme di pagine, che ora è visto come lo spazio di indirizzamento di un processo, viene interpretato come un insieme di segmenti di memoria). Per accedere ad un dato segmento, si specifica quindi la sua pagina, il suo indirizzo all'interno della pagina e l'offsett all'interno del segmento. Questa tecnica è efficace in quanto permette addirittura di condividere informazioni tra processi. In effetti, che bisogno c'è di ricopiare negli spazi di indirizzamento informazioni che devono solo essere lette? Per esempio, vi è una zona di dati a cui tutti i processi (eventualmente con i loro figli ed i thread) accedono in sola lettura. Ricopiare tali informazioni in ogni Address Space è un enorme spreco di memoria. L'approccio della segmentazione permette di condividere tali informazioni.

In effetti, è ciò che facciamo quando usiamo mmap con il flag MAP_SHARED. Stiamo dicendo di rendere i segmenti che compongono quelle pagine condivisi e non più copy-on-write.

Per quanto riguarda il problema:

1) Essendo la paginazione a due livelli, abbiamo bisogno di un indirizzo per decretare la entry sulla prima tabella ed un indirizzo per le entry sulla seconda tabella. Gli indirizzi sono a 9 bit, quindi in totale 18 bit di indirizzo e 22 bit di offset che fanno un indirizzo di 40 bit.
2) Delle 10 tabelle disponibili, la prima deve essere di primo livello e non punta ad alcun frame, mentre abbiamo disponibili altre 9 tabelle di primo livello, per un totale di 9*512 frames correntemente occupati. Si noti che so con precisione il numero delle pagine di secondo livello in quanto ogni tabella ha 9 entry (9 indirizzi), altrimenti non avrei potuto dirlo.

DOMANDA 2
Si consideri un insieme di 3 processi {LETT, PROD1, PROD2}. LETT accede periodicamente ad un segmento di memoria condivisa M fatto di due slot (M[1] ed M[2]) per leggerne il contenuto. PROD1 e PROD2 aggiornano periodicamente il contenuto della memora condivisa secondo il seguente schema: PROD1 aggiorna M[1] mentre PROD2 aggiorna M[2]. Lettura ed aggiornamento devono avvenire in mutua esclusione. L'aggiornamento di entrambi gli slot M[1] ed M[2] deve figurare come se fosse un'azione atomica, ovvero LETT non può leggere il contenuto della memoria condivisa M nel caso in cui sia stato aggiornato solo uno dei due slot. Inoltre, la lettura del contenuto della memoria condivisa da parte di LETT non deve essere impedita nel caso in cui solo uno tra i due processi PROD1 e PROD2 sia pronto per l'aggiornamento. Si risolva tale problema di sincronizzazione, utilizzando solo semafori, fornendo lo pseudo-codice delle procedure: AGGIORNA (usata da PROD1 e PROD2) e LEGGI (usata da LETT).

Ho bisogno dei seguenti semafori (soluzione non corretta, verrà spiegato in seguito perché):

1) Writers: {2 entry con 1 l'una token inizialmente disponibili}-> ogni processo scrittore preleverà un token per accedere in scrittura.
2) Reader: {1 entry con 2 token inizialmente non disponibili};

AGGIORNA:

WAIT(Writers, entry i-esima);
<Deposita il messaggio sul frammento i-esimo di memoria condivisa>;
SIGNAL(Reader, 1);

LEGGI:
WAIT(Reader, 2);
<Leggi tutto il contenuto della memoria condivisa M>
SIGNAL(Writers, tutte le entry);

NOTA: questa soluzione purtroppo non è corretta perché non prende in considerazione alla lettera quanto richiesto nelle righe evidenziate in giallo. Infatti, esse chiedono che LETT possa leggere indefinitamente la memoria anche se non vi sono nuove istruzioni da consumare. Ergo, potenzialmente LETT deve poter leggere all'infinito la stessa informazione anche se gli altri due non sono pronti a scrivere. Questo non l'avevo preso in considerazione, perché sinceramente sembrava una cosa molto sommaria e di dubbia realizzazione in un efficiente sistema, ma data anche la soluzione ufficiale presente sul sito del Professore sembrerebbe essere questa la realizzazione richiesta. Ciò che non mi convince è che questa soluzione si avvale di due funzioni per i processi scrittori quando nel testo ne viene chiesta una da usare per entrambi. In ogni caso, questa è la mia soluzione. La differenza con quella sul sito del Prof è che io non mi avvalgo di P3, perché ritengo che alla fine il codice sia identico:

AGGIORNA1:
SIGNAL(P1);
WAIT(P2);
WAIT(S, 1);
<Deposita il messaggio sull'area di memoria>;
SIGNAL(S, 1);

AGGIORNA2:
SIGNAL(P2);
WAIT(P1);
WAIT(S, 1);
<Deposita il messaggio sull'area di memoria>;
SIGNAL(S, 1);

LEGGI:
WAIT(S, 2);
<Leggi>
SIGNAL(S, 2);

Essenzialmente il processo lettore inserisce gettoni che può potenzialmente subito ritirare se entrambi i semafori non sono subito pronti. Infatti, all'inizio abbiamo quello che sembra un vero e proprio "handshake" tra P1 e P2. Essenzialmente, finché non sono ambedue pronti in questo modo il lettore può comunque leggere, perché entrambi i processi possono iniziare a scrivere *solo ed unicamente* quando si sbloccano a vicenda.

2020-06-18:

DOMANDA 1
Descrivere l'algoritmo di di scheduling di CPU Round-Robin (RR), discutendo l'impatto della scelta del time-slice sul comportamento dei processi I/O bound. Si consideri inoltre il caso in cui vengono attivati allo stesso istante temporale 5 processi P1, ... , P5. P1 e' un processo I/O bound che interagisce con un dispositivo D per N=10 volte e poi termina (il processo termina non appena l'ultima interazione e' completata). Tutti gli altri processi sono CPU bound di durata infinita. Supponendo che (1) la lunghezza di ogni CPU burst di P1 prima di una nuova richiesta di I/O sia nota a priori e pari a 5 ms., (2) il dispositivo D impieghi 9 ms. per ogni interazione, (3) il tempo di CPU per il dispatcher RR sia nullo, si determini (motivando la risposta) quale tra i seguenti valori del time-slice minimizza il tempo di attesa di P1 in caso di dispatching RR: 2 ms., 6 ms., 18 ms.
Si calcoli inoltre il tempo di attesa almeno nel caso del time-slice che minimizza tale tempo.

Il Round Robin è stato già ampiamente discusso.
Per quanto concerne il problema, essenzialmente si ha che il quantum slice non può essere troppo piccolo (altrimenti il processo spenderebbe molto tempo bloccato in richiesta di I|O) ma al contempo non può essere neppure troppo lungo, perché altrimenti si perderebbe del tempo. Il processo in tutto ha bisogno (in un solo ciclo) di 5+9= 14ms. In quanto tale:

Con un time slice di 6 ms si ha:
(1, tutto il CPU burst, poi si perde 1 ms di I|O perché lui viene deschedulato (ha finito il cpu burst) -> (6ms per P2) -> (3 ms in P3 -> fine dell'I|O, P1 chiede di essere rischedulato -> ultimi 3 ms di P3) -> (12 ms P4, P5) -> (6ms di P2 – nota, P2 è stato schedulato PRIMA di P1 perché P1 ha terminato la sua chiamata bloccante durante il time slice di P3) -> P1

Il tempo di attesa è definito come il tempo che intercorre dal momento in cui sono ready al momento in cui entro in CPU. Si hanno quindi 3 (in attesa di P3) + (6*3) (in attesa di P4, 5, 2)= 24ms di tempo di attesa.
Ripetendo il calcolo per 2, 18 si vede che i tempi di attesa sono maggiori. In realtà, la risposta era estremamente intuitiva: con 2ms ho bisogno di 3 cicli solamente per fare il CPU burst mentre con 18 sulla prima richiesta spreco 13 ms.

DOMANDA 2
Descrivere i possibili metodi di indirizzamento nella tecnica di comunicazione basata su scambio di messaggi.
Indicare inoltre che tipologia di indirizzamento viene utilizzata sui sistemi UNIX e Windows.

Tecniche di indirizzamento:
Diretta: essenzialmente è una comunicazione end to end tra due entità, che non si avvale di alcun intermediario;
Indiretta: ci si avvale di un buffer come intermediario tra le due entità in comunicazione. Vi sono dei problemi legati a questo espediente: la persistenza del buffer, per esempio. Ci sono alcune casistiche in cui non è importante che tutti i messaggi vengano letti, e quindi il buffer può cancellare dati quando si riempie, altri che si basano su sistemi di controllo per fermare il sender. Inoltre, a seconda delle applicazioni, l'ordine di arrivo dei messaggi può o meno essere mantenuto.
Per quanto concerne il buffer, ci sono due modi principali di allocarlo:

1) Kernel: in tal caso il ricettore non deve impostare una receive(), e la dimensione del buffer può essere scelta tra nulla (1 solo messaggio, tipica dei roundes-vouz) limitata ed illimitata;
2) Memoria user: in tal caso si deve impostare una receive, pena la possibilità di perdere il messaggio.

UNIX: utilizza sia send che receive sincrone (bloccanti e non), con un buffer di dimensione limitata livello kernel identificato da un nome che non è di file system (leggasi, un nome come potrebbe essere quello di un file), bensì un codice numerico identificativo

per permettere a threads e processi di accedere a questa struttura. Palesemente l'indirizzamento è indiretto. Supporta la gestione del tipo di messaggi, consentendo il multiplexing.

Windows: utilizza un buffer livello kernel di dimensione limitata, receive e send sia sincrone che asincrone, bloccanti e non. Il resto è identico a UNIX, ma dato che non viene fatta gestione della tipologia di messaggi non è possibile il multiplexing.

Sebbene non richiesto nella domanda, qui di seguito si illustrano le chiamate sincrone ed asincrone:
Primitive di:
SPEDIZIONE: si identificano primitive

1) (Non) bloccanti: sincrona, randez-vous. La sincrona prevede di cedere il controllo solo quando non vi è rischio di danneggiare l'informazione nel buffer; randez-vous in genere si avvale di buffer kernel di ampiezza nulla (un solo messaggio) e prevede l'uso di ack per capire se il messaggio sia o meno arrivato a destinazione;
2) Bloccante: asincrona: non si cura della safety del messaggio e potenzialmente potrebbe anche sovrascriverlo pur di riutilizzare subito il buffer;

RICEZIONE: si identificano due primitive ambedue (a)sincrone: la bloccante mette in WAIT un processo sinché non vi è almeno un'informazione da leggere nel buffer, mentre la non bloccante fa palesemente l'opposto, non curandosi della disponibilità o meno dell'informazione nel buffer.

2020-09-17:

DOMANDA 1
Descrivere le caratteristiche salienti di un file system UNIX, indicando anche i relativi aspetti di sicurezza.

Abbiamo già parlato del File System UNIX. Per quanto concerne gli aspetti di sicurezza, ci viene chiesto di parlare delle ACL. Queste vengono implementate in maniera ingegnosa mediante un "file shadow" accoppiato al file originario. Su tale File Shadow vengono depositate le informazioni per la sicurezza, come quali utenti possono accedere al file oppure quali gruppi, chi ha permessi di esecuzioni e chi di lettura e chi di scrittura e così via. Ovviamente, essendo un file esso ha bisogno di un I-node per essere identificato nel Virtual File System. Tale I-node è detto "shadow" ed è collegato all'I-node del file originario. Quindi ogni qualvolta un utente o un gruppo volessero accedere al file in questione, tramite il link che persiste tra i due I-node viene consultato il file shadow.

DOMANDA 2
Si consideri un insieme di processi (P1 .. PN) e un ulteriore processo PROC. Si consideri inoltre una memoria condivisa M. Il processo PROC scrive periodicamente un nuovo messaggio su M, che deve essere letto da N/2 tra i processi Pi. Data una coppia di messaggi successivi m e m' scritti da PROC, m' non puo' essere letto da un processo Pi se questo processo aveva letto il messaggio m. Inoltre PROC non puo' depositare su M un nuovo messaggio se l'ultimo messaggio da lui depositato non sia stato gia' letto da N/2 tra i processi Pi. Si fornisca la soluzione di tale problema di sincronizzazione, utilizzando solo semafori, fornendo lo pseudocodice delle procedure SCRIVI e LEGGI usate, rispettivamente, da PROC e dai processi Pi.

Come in un problema precedentemente affrontato, abbiamo due tipi di lettori: i *primi lettori* ed i *secondi lettori*. Anche in questo caso, la scrittura *periodica* del processo PROD impone che si debba utilizzare una sola funzione per ambedue le scritture, delegando la complessità alla funzione di lettura. In particolare, ho bisogno di una variabile globale che chiameremo *phase* che stia ad indicare la *generazione* di lettori. La funzione del processo scrittore dovrà essere il più generale possibile, sicché possa essere utilizzata da ambo le generazioni di lettori. Veniamo quindi alla soluzione, avvalendoci dei seguenti semafori:

1) Writer: {1 entry con N/2 token inizialmente disponibili} -> Segnala al writer che N/2 processi hanno finito di leggere;
2) Readers: {N entry con inizialmente 0 token disponibili, li inserirà periodicamente il processo scrittore};
3) Reader-first: {1 entry con inizialmente zero token, ma PROD ne aggiungerà N/2 all'occorrenza};

global: int phase= 0;
LEGGI:
WAIT(Readers-First, 1);
WAIT(Readers, entry i-esima);
<Leggi l'informazione>;
SIGNAL(Writer, 1);

SCRIVI:

```
WAIT(Writer, N/2 token);
<Scrivi l'informazione su memoria condivisa>;
SIGNAL(Reader-first, N/2);

if(phase == 0){
        int i= 0;
        for(i from 0 to N (N non compreso))
                SIGNAL(Readers, entry i-esima);
}
phase= (phase + 1)%2.
```

Quel che si è fatto è in realtà un trucchetto interessante, ovvero, ogni volta che phase è zero (quindi una volta sì ed una no) PROD, dopo aver comunque scritto un messaggio, farà una segnalazione su tutti i semafori lettori, e poi sul semaforo conteso che determina i lettori di prima generazione. Phase viene settato poi a 1, e la seconda volta che PROD può scrivere, tutti i lettori di prima generazione avranno terminato la lettura, quindi gli indici prima messi a disposizione nel semaforo Readers sono alla portata di tutti coloro che che sono di seconda generazione, che si possono ora aggiudicare un token del semaforo Reader-First, N/2);

Notare che non ho bisogno di un semaforo per impedire che un lettore legga due volte la memoria condivisa, in quanto ho la certezza che tutti i secondi lettori sono in fervente attesa di un token su Reader-first, quindi tutti si aggiudicano quei token e nessuno ne prende più di uno.

2020-09-17:

DOMANDA 1
Descrivere gli obiettivi e le caratteristiche delle tecniche di gestione del "resident-set" nei sistemi operativi basati su memoria virtuale. Descrivere inoltre l'algoritmo dell'orologio per la sostituzione delle pagine.

Il Resident Set indica quante pagine logiche l'app può mantenere contemporaneamente in RAM. Ovviamente la dimensione del residente set è cruciale nella stima dei page fault, perché un RS troppo piccolo potrebbe portare ad un numero inaccettabile di page fault. Tuttavia, un RS troppo grande potrebbe privare di spazio altre applicazioni ed ottenere lo stesso effetto di causare molti page fault per suddette applicazioni.

L'allocazione del Resident Set può quindi essere:

1) Fissa: viene decisa una taglia iniziale all'attivazione del processo; se devo scegliere una vittima, la posso scegliere solo tra le mie pagine;
2) Variabile: il numero dei frames può variare per ogni processo, in quanto diviene possibile scegliere le vittime anche tra le pagine di altre applicazioni;
3) Mista: si hanno le stesse caratteristiche dell'allocazione fissa, con l'unica differenza che periodicamente viene rivalutata la taglia del resident set in base a due parametri, il Working Set e la frequenza di page fault.

(Sebbene non richiesto, si spiega anche il working set):
Il Working Set è essenzialmente un insieme in cui sono contenute le D pagine più utilizzate di recente. Se una pagina è quindi nel working set, è probabile che recentemente sia stata utilizzata. Il Resident Set è auspicabilmente di taglia pari al working set, in modo tale da rendere ottimale il numero di page fault. Infatti, il WS si basa sull'immediato passato per "prevedere" la futura località del processo. In quanto tale, se D è troppo grande potrebbe essere presa una zona troppo vasta e la stima non sarebbe ottimale, ovvero si cattura anche la *variazione di località*, mentre per una stima troppo bassa potrebbe non essere catturata tutta la località.

(Si parla infine anche del Thrashing in modo da concludere il discorso):
Essenzialmente è un fenomeno per cui, nonostante (idealmente) all'aumentare del grado di multiprogrammazione (leggasi: del numero di threads/processi in contesa per la CPU) dovrebbe aumentare il throughput, si arriva ad un punto in cui ci sono troppi processi in memoria di lavoro, e quindi i RS sono obbligati a "restringersi", portando ad un aumento improponibile del numero di page fault, ergo il throughput, e quindi di conseguenza l'utilizzo della CPU, diminuisce, mentre aumenta il numero di accessi su memoria di massa -> possibile usura del dispositivo.

DOMANDA 2
Sia dato un insieme processi (P1, ... , Pn) con n=16. Ogni processo Pi deposita periodicamente un nuovo messaggio sull'i-esimo slot di una memoria condivisa M di n slot. Un ulteriore processo LETT legge i messaggi accedendo ad M secondo la regola del buffer circolare. Ogni 4 nuovi messaggi letti, LETT inserisce una risposta in un ulteriore

buffer di memoria condivisa R a slot singolo, destinata a tutti i processi Pi scrittori dei 4 messaggi appena letti da LETT. Pi rimane in attesa della risposta al suo messaggio dopo averlo inserito nell'apposito slot di M. LETT deve entrare in stato di attesa nel caso in cui lo slot della memoria condivisa correntemente di interesse non contiene alcun nuovo messaggio, oppure quando il buffer R contiene una risposta non ancora letta dal relativo destinatario. Si risolva tale problema di sincronizzazione, usando solo semafori, fornendo lo pseudo-codice delle procedure LEGGI e PRODUCI usate rispettivamente da LETT e Pi.

Non viene data alcuna informazione esplicita sull'ordine di scrittura, ma viene detto che i processi Pi depositano *periodicamente* il messaggio, quindi fa pensare a qualcosa che accade ad intervalli di tempo costanti ed in quanto tale cerchiamo un ordine in scrittura. Allo stesso tempo, il lettore accede sincronizzandosi sulla entry dell'informazione desiderata. Ogni 4 informazioni lette, il processo deve bloccarsi, ma questo non significa che gli altri processi Pi non possano scrivere. Ho quindi bisogno dei seguenti semafori:

1) Writers: {N entry, 1 token l'una inizialmente disponibile};
2) Reader: {N entry con 0 token inizialmente disponibili; vengono caricati mano a mano dai processi scrittori};
3) Done: {1 entry con 0 token, ma verrà portata a 4} -> Serve per segnalare a LETT che tutti i 4 processi Pi hanno terminato la lettura della risposta nella memoria condivisa R;
4) Available: {N entry con 0 token inizialmente disponibili, progressivamente portate ad 1} -> LETT segnala che c'è qualcosa da leggere; non faccio un semaforo con 1 sola entry e 4 token perché altrimenti processi "vecchi" potrebbero riappropriarsi della risposta;

SCRIVI:
```
WAIT(Writers, entry i-esima);
<Scrivi sul frammento i-esimo di memoria>;
// Mettiti in attesa della disponibilità della risposta:
WAIT(Available, entry i-esima);
<Leggi la risposta da R>;
SIGNAL(Done, 1);
```

LEGGI:
```
int i= 0;
for(i from 0 to N (N non compreso)){
        WAIT(Reader, entry i-esima);
        <Leggi il contenuto dalla memoria i-esima>;
        // Segnala al processo Pi che può adesso tornare a scrivere:
        SIGNAL(Writers, entry i-esima);
        // Controlla se hai già fatto 4 letture:
        if((i-1)%4 == 0){
                WAIT(Done, 4);
                <Scrivi il messaggio su memoria condivisa R>;
                SIGNAL(Available, entry i-3, i2, i-1, i);
}
```

2021-20-01:

DOMANDA 1
Descrivere le problematiche di scheduling dell'I/O su hard-disk a rotazione, e gli algoritmi che il sistema operativo ha utilizzato storicamente per affrontare tali problematiche.

Una corretta schedulazione delle informazioni con cui interagire su disco è fondamentale per evitarne l'usura. I dischi principalmente utilizzati oggi sono due:

1) Dischi magnetici a rotazione, che si avvalgono di tracce concentriche sulle quali scorre una testina meccanica rotante in grado di leggere ed eventualmente modificare informazioni. La loro problematica è legata per l'appunto alla natura prettamente elettro-meccanica. Infatti, ripetuti spostamenti della testina ne usurano meccanicamente la struttura, determinando quindi un calo di efficacia con l'avanzare del tempo; questa tecnologia è attualmente la più diffusa grazie al conveniente rapporto costo/benefici, sebbene la velocità non sia ottimale e si possa solo contemporaneamente leggere o scrivere una singola traccia. Tuttavia, richieste sulla stessa traccia possono essere servite simultaneamente.

2) SSD: sono dispositivi puramente elettrici che si avvalgono di zone di memoria chiamate “blocchi fisici” in cui si possono depositare dei “blocchi logici” anche detti “pagine” (da non confondersi con le pagine di memoria delle memorie di lavoro). La loro velocità è maggiore degli HDD, tuttavia hanno un problema invalidante: nonostante lettura e scrittura contemporanee di un blocco siano effettivamente possibili, la scrittura è realizzabile solo cancellando il blocco e riscrivendo l’informazione aggiornata in esso. Ogni volta che ciò avviene, il blocco si usura e si è stimato che, in media, dopo 100’000 cancellazioni il blocco diventi inutilizzabile. Per ovviare a tale problema si utilizza un algoritmo detto “di fusion”, ovvero si cerca di raggruppare istruzioni dirette alla stessa traccia in modo da determinarne il risultato finale e svolgere meno operazioni di scrittura possibile su tale blocco. Inoltre, per aggiornare un’informazione presente in memoria senza tuttavia cancellare il contenuto precedente, ci si avvale di un buffer cache in cui si deposita l’informazione, la si aggiorna, poi si cancella il blocco e l’informazione viene riscritta su di esso.

Per quanto concerne gli algoritmi utilizzati, l’evoluzione parte dal più basico, ovvero First Come First Served, che prevede di soddisfare le richieste per tracce (ora ci riferiamo ad algoritmi di schedulazione di dischi meccanici) secondo l’ordine di arrivo. Ciò ovviamente non minimizza né il tempo di ricerca né il numero di spostamenti, ed è quindi pericoloso per gli HDD; la successiva evoluzione che porta allo Shortest Service Time First fa si che ad essere soddisfatta sia quella richiesta che dista di meno dall’attuale traccia sulla quale mi trovo. Al contrario del suo predecessore può produrre starvation; infatti, se continuano ad arrivare richieste su tracce localmente vicine a dove si trova la testina (palesemente possibile nel caso di accesso a zone contigue dovuto al fenomeno della località) le richieste più lontane vengono messe in secondo piano. Quindi da una parte salvaguarda il disco, ma dall’altra non riduce il seek time e può provocare starvation.

La prima vera rivoluzione arriva con l’algoritmo dell’ascensore: CSCAN si muove in una data direzione fino al termine delle tracce o delle richieste, e poi si gira e ricomincia nell’altro verso. Sebbene risolva parecchi problemi, può provocare starvation di processi più “antichi”. Si pensi infatti di aver appena servito una richiesta sulla prima traccia e di essersi spostati sulla 2, e poi arrivino una o più richieste sulla 1: dato che la testina dovrà andare tutto in avanti e poi ritornare indietro, questo provoca starvation di queste richieste. La versione successiva, C-SCAN si sposta solo in una direzione, e, una volta che ciò è finito, torna indietro ripartendo dalla prima traccia. Ovviamente questo risolve il problema di starvation di richieste più “antiche”, tuttavia penalizza quelle che avvengono nelle tracce più lontane nelle quali la testina è appena passata.

Attualmente viene utilizzato l’algoritmo detto FSCAN; non è propriamente un algoritmo, quanto un modo di avvalersi di quanto sinora illustrato senza incorrere nei problemi di starvation derivanti da talune situazioni patologiche che obbligano la testina a rimanere in una data zona. FSCAN si avvale infatti di due code distinte, di cui la prima è la coda di input, in cui la gestione è first come first served. Poi queste richieste passano in una seconda coda dopo essere state riordinate in base ad una precisa policy (tra quelle prima illustrate), questa coda, detta di scheduling, assume un preciso numero di richieste e basta, e finché quelle non sono state soddisfatte non possono esserne accettate altre.

Sebbene non richiesto, ritengo sia istruttivo spiegare anche il problema della consistenza nei file system: dopo uno shutdown infatti le informazioni ancora pendenti in buffer di I|O potrebbero essere perdute. Si può *ricostruire* il file system mediante l’analisi di MFT (Windows) e vettore di I-nodes (UNIX). Si tengono due bit mask, che identificano se un blocco è o meno occupato. Se ambedue recitano 0, il blocco viene settato a disponibile perché eventualmente c’è stato un errore. Se invece un blocco compare due volte libero, viene decrementata l’occorrenza. Se il blocco è invece utilizzato due volte, questo significa che il blocco compare in due file, e quindi viene duplicato e sostituito in uno dei file (infatti, a livello hard disk non ha assolutamente senso duplicare informazioni -in termini di blocchi- comuni a più file. Se ci sono blocchi comuni a più file allora tutti essi attingeranno da lì un po’ come si fa con la segmentazione nella paginazione segmentata, e finché nessuno manifesta il desiderio di scrivere tali blocchi rimangono “copy on write”).

Infine, dato che ogni tanto queste domande capitano, si elencano le facilities per sincronizzare i file systems:
UNIX: fsync-> Copia tutte le informazioni di un file su disco rigido, attendendo conferma che le informazioni siano sicure, e aggiorna anche i metadati; fdatasync -> come fsync, ma non si occupa dei metadati;

WINDOWS: FlushFileBuffer permette di riversare il contenuto bufferizzato di un dato file su disco rigido.

DOMANDA 2
Si consideri un insieme di processi {P1,...,Pn} che periodicamente cercano di leggere un nuovo messaggio. Ciascun processo Pi cerca di leggere il nuovo messaggio da una memoria condivisa M, e rimane bloccato in attesa che esso venga scritto su M se non presente. Un ulteriore processo PROD deposita un nuovo messaggio su M periodicamente, solo se almeno n/2 processi sono in attesa di riceverlo, altrimenti rimane in attesa che tale condizione sia verificata. Inoltre, PROD puo' scrivere un nuovo messaggio su M solo dopo che il suo precedente messaggio e' stato letto da tutti i processi nell'insieme {P1,...,Pn}. Risolvere tale problema di sincronizzazione, utilizzando solo semafori, fornendo lo pseudo-codice delle procedure LEGGI e SCRIVI utilizzate, rispettivamente, dal generico processo Pi e da PROD.

Il problema qui è come indicare “almeno N/2 processi devono aver manifestato l’attenzione di leggere tale informazione”; si può utilizzare un semaforo con N/2 token e lasciare che PROD faccia la WAIT su di esso, ma questo significa che sicuramente più di N/2 processi manifesteranno l’intenzione di leggere l’informazione. Di conseguenza, PROD fa una WAIT di N/2 token, ed i restanti

token rimangono inseriti nel semaforo rimarrebbero poi per il "prossimo giro", ingannando quindi PROD. Le cose sono due: o si trova un'altra soluzione, oppure si usa l'equivalente di una SEMCTL che, in linea di massima, non dovrebbe essere impedito dal problema (in fondo, si chiede di utilizzare semafori…). Prendiamo il caso peggiore e quindi non usiamo la SEMCTL. In tal caso dobbiamo badare ai gettoni rimanenti. Possiamo utilizzare una variabile globale che PROD imposta ad 1 quando ha raggiunto N/2 gettoni (ed ha quindi passato la WAIT) eventualmente impedendo ad ulteriori lettori di fare la SIGNAL sul semaforo che gli permette di scrivere.

Ho bisogno dei seguenti semafori:

1) Go: {1 entry, N/2 token inizialmente non disponibili};
2) Writer: {1 entry, N token inizialmente disponibili};
3) Readers: {N entry, inizialmente con 0 token disponibili} -> PROD segnala al generico processo che è presente l'informazione. Singolo così da evitare che un processo prenda più token

Globale: j= 0;

SCRIVI:
```
WAIT(Go, N/2);
j= 1;
WAIT(Writer, N);
// Tutti hanno letto: scrivi un nuovo messaggio:
<Scrivi un messaggio su M>;
j= 0;

for(i from 0 to N)
        SIGNAL(Readers, entry i-esima);
```

LEGGI:

```
if(j == 0)
        SIGNAL(Go, 1);

// Altrimenti significa che non c'è bisogno del mio token, perché PROD ha già avuto un numero sufficiente di token;
WAIT(Readers, entry i-esima);
<Leggi il contenuto della memoria i-esima>;
SIGNAL(Writer, 1);
```

Notare che anche se adesso un processo tornasse indietro all'inizio del codice e aggiungesse un token a Go, questo token sarebbe relativo alla volontà di voler recepire il prossimo messaggio e, finché PROD non sbloccherà di nuovo la sua entry di semaforo, egli non potrà accedere alla memoria. Notare inoltre che non vi è pericolo che un processo, una volta terminata la lettura, rientri nel codice trovando j == 1 (o, almeno, non è possibile finché non è PROD ad impostarlo perché ha ricevuto N/2 token) in quanto j viene settato a zero non appena scritto il messaggio.

2021-02-18:

DOMANDA 1
Descrivere l'algoritmo dell'orologio per la selezione della pagina vittima in sistemi operativi che supportano la memoria virtuale.

Già ampiamente discusso nel resto del file (consiglio vivamente di recuperarsi la *domanda 3* del *2019-09-16* in quanto vi sono considerazioni interessanti anche circa l'anomalia di Belady -qui non richiesta, ma è sempre un punto in più per voi all'esame se ne parlate!-);

DOMANDA 2
Si consideri un insieme di processi {P1,...,Pn} che periodicamente scrivono un nuovo messaggio su una memoria condivisa M. Un ulteriore processo LETT legge un nuovo messaggio da M periodicamente rimanendo bloccato in attesa del messaggio qualora non fosse stato scritto. Ogni processo Pi che tenta di scrivere un nuovo messaggio rimane anche esso bloccato in attesa che l'ultimo messaggio scritto su M sia stato letto. Inoltre, un ulterore processo RAND, assegna la possibilita' di scrivere un nuovo messaggio ad un processo Pi scelto a caso, solo dopo che l'ultimo processo precedentemente scelto abbia depositato il proprio messaggio. Risolvere tale problema di sincronizzazione, utilizzando

solo semafori, fornendo lo pseudo-codice delle procedure SCRIVI, LEGGI e SELEZIONA utilizzate, rispettivamente, dal generico processo Pi da LETT e da RAND.

Lo "scegliere a caso" di RAND si traduce in un semaforo "Competition" con 1 entry e 1 solo token per il quale i vari processi competono. Ho bisogno dei seguenti semafori:

1) Competition: {1 entry, 0 token inizialmente disponibili};
2) Go: {1 entry, 0 token inizialmente disponibili} -> Utilizzato da LETT per far sapere che ha effettivamente letto il messaggio e che un nuovo processo può scrivere sulla memoria;
3) Done: {1 entry, 1 token inizialmente disponibili} -> Con questo messaggio il generico processo scrittore dice di aver finito di scrivere il messaggio su memoria condivisa.
4) Reader: {1 entry, 0 token inizialmente disponibili} -> utilizzato per segnalare al processo lettore che vi è una nuova informazione da consumare;

SELEZIONA:
WAIT(Done, 1);
// Scegli un processo a caso mettendo un token in Competition:
SIGNAL(Competition, 1);

SCRIVI:
// Competi per la scrittura:
WAIT(Competition, 1);
// Aspetta che ti sia dato il permesso da LETT di leggere:
WAIT(Go, 1);
<Scrivi un nuovo messaggio sulla memoria M>;
SIGNAL(Done, 1);
SIGNAL(Reader, 1);

LEGGI:
WAIT(Reader, 1);
<Leggi il messaggio da memoria condivisa M>;
// Segnala che hai finito di leggere:
SIGNAL(Go, 1);

E con questo si conclude questo "eserciziario", se così mi è concesso chiamarlo. In bocca al lupo per l'esame!