Sistemi operativi per il corso di diploma universitario in Ingegneria Elettronica anno accademico 2000/01

Agenda delle lezioni dell'A.A. 2000/01

Posizione originaria: <URL:http://fly.cnuce.cnr.it/didattica/agenda00.html>.

Abbreviazioni usate nell'agenda:

:2L
: 2 ore di lezione (sviluppo di nuovi argomenti)

:1E
: 1 ora di esercitazioni (esercizi in classe)

:1S
: 1 ora di laboratorio (pratica in classe o in laboratorio)

S:7.6; P: 2.1-2.12
: Argomenti trattati nel testo di riferimento Silberschatz, capitolo 7.6, e nel manuale pSOS, da pagina 2-1 a pagina 2-12.

Cos'è un sistema operativo

S: 1-3

- [1:2L] Struttura a strati della macchina, processore, microprogramma, linguaggio macchina, sistema operativo, librerie, ambienti, programmi.
- S.O. come macchina virtuale estesa, che presenta un'interfaccia uniforme e semplice a diversi dispositivi (p.es. accesso al disco).
- S.O. come gestore delle risorse, che ne consente l'uso coordinato a più programmi, anche contemporaneamente (p.es. uso della stampante).
- Sistema monoprogrammato, multiprogrammato, prelazione e cooperazione, condivisione di tempo, real-time hard e soft.
- Protezione hardware (modalità protetta), spazio utente e supervisore, istruzioni privilegiate, trap, interrupt, system call, eccezioni.
- [2:1L] Processi: nascita, esecuzione, morte, figli, diritti. Meccanismi di comunicazione sincroni: messaggi, memoria condivisa, semafori. Meccanismi di comunicazione asincroni: segnali, mascheramento.
- Memoria: gerarchie di memorizzazione, prezzo, velocità, dimensioni, volatilità. Gerarchie di memoria per la memoria centrale: cache di primo e secondo livello, RAM, spool su disco. Per la memoria secondaria: disco, nastri, batterie di nastri o CD.
- File: directory, working directory, nomi assouti e relativi, apertura, chiusura, accessi, diritti. Significati di filesystem: struttura logica dell'insieme dei file, sottosistema del kernel che la implementa e gestisce, l'insieme di tutti i file accessibili da una macchina, la gerarchia di file che è allocata su una partizione.
- [3:2L] Kernel monolitici e modulari (Unix), a strati protetti con supporto hardware (Multics),

macchine virtuali pure (VM), microkernel (Mach). Macchine virtuali vecchio stile (IBM 370), emulatori di macchine (Vmware), emulatori di processore (Virtualpc), macchine virtuali che non emulano alcuna macchina fisica (Java).
- Sistema monoprocessore, multiprocessore, sistemi ridondanti, sistemi distribuiti.
- Shell, sistema a finestre. Programmi che accompagnano il sistema operativo. Interfaccia utente, interfaccia di programmazione. Solo meccanismi (DOS), solo politica (Macintosh).

Boot

S: 3.8, 13.3.2

- [4:2L] Processo di boot da ROM. Alcune alternative possibili:
  - Singolo programma in ROM che viene eseguito all'accensione.
  - Piccolissimo programma caricatore, che copia in RAM il sistema contenuto in ROM e poi lo esegue facendo un salto.
  - Boot a più stadi.
- Il boot a più stadi è la norma nelle macchine il cui sistema operativo risiede su disco (DOS = disk operating system). Nelle architetture di tipo PC avviene questa sequenza di passi:
  - Viene dato il controllo al *POST* (power on self test).
  - Il controllo passa al BIOS, che verifica su quale dei possibili dispositivi di boot è presente un sistema operativo.
  - Nel caso in cui trovi un *MBR* (master boot record) su disco rigido lo legge. Si tratta del primo settore del disco rigido. Questo contiene la tabella delle partizioni ed un caricatore.
  - Il caricatore legge la tabella delle partizioni, determina la partizione attiva, e ne legge il primo settore usando le chiamate del BIOS.
  - Il settore letto contiene il *boot record*, che è un caricatore. Questo, usando le chiamate del BIOS, legge da una posizione fissa il kernel del sistema operativo e gli dà il controllo.
  - Il kernel, dopo aver inizializzato le periferiche, crea il primo processo, che in Unix si chiama *init*.
  - Questo processo è il padre di tutti gli altri processi. In particolare è il padre del processo di *shell*, che costituisce l'interfaccia utente.

Dal sorgente all'esecuzione: l'odissea di un programma

S: 8.1…

- Il *compilatore* in senso stretto è il programma che legge un sorgente e produce un *oggetto*, cioè un file che contiene codice in linguaggio macchina e informazioni di controllo.
- Il processo più semplice di passaggio dal sorgente al codice eseguibile è quello eseguito dai compilatori di tipo *compile-and-go*, come ad esempio il Turbo Pascal con cui nacque la Borland negli anni '80. Il compilatore legge il codice sorgente, produce un eseguibile in memoria e gli dà il controllo. Il codice prodotto è posto direttamente nella memoria da dove viene eseguito.
- Il passo successivo di complicazione è quello che porta da un codice sorgente ad un programma monolitico da porre in ROM. Si usa solo per programmi estremamente semplici. Il compilatore genera un oggetto non rilocabile che viene bruciato direttamente in ROM, ad una locazione fissa.
- Più in generale, si parte da più file sorgenti, ognuno dei quali viene compilato producendo *file oggetto*. Più file oggetto possono essere archiviati assieme in un unico file detto *libreria*. I vari oggetti sono quindi collegati assieme da un *linker*, che produce un *file eseguibile*. Quest'ultimo è quindi caricato in memoria ed eseguito.
- Il file eseguibile può essere di diversi tipi:

  - *Oggetto con riferimenti assoluti*: l'oggetto contiene informazioni di controllo che indicano l'indirizzo in memoria al quale va caricato, oppure questa informazione è nota in qualche altro modo. Un oggetto di questo genere viene ad esempio prodotto per essere bruciato in una ROM.
  - *Oggetto rilocabile*: l'oggetto contiene almeno un riferimento assoluto. Tuttavia può essere collocato in una posizione arbitraria di memoria da un *caricatore rilocante*. Questo usa le informazioni di controllo contenute nella *tabella di rilocazione* dentro il file oggetto per aggiustare i riferimenti assoluti. Un esempio di questo tipo di oggetto sono i file .EXE del sistema operativo MS-DOS.
  - *Codice indipendente dalla posizione*: l'oggetto contiene solo *riferimenti relativi*, per cui può essere posto in una posizione arbitraria della memoria senza necessità di rilocazione. È il caso dei file .COM del sistema operativo MS-DOS.
- La tabella di rilocazione contiene voci di rilocazione (*relocation entry*). Per ogni voce ci sono dei campi che specificano il tipo di voce, la posizione all'interno dell'oggetto, ed il valore da sommare a quella posizione. Sia la posizione che il valore sono indirizzi di memoria.
  - Il *tipo* di voce è ad esempio «indirizzo a 32 bit di tipo little endian», che indica su quanti byte e in che modo va scritto il valore rilocato.
  - La *posizione* della voce è un *offset*, cioè una distanza da un punto fisso, per esempio l'inizio del file oggetto. Indica la posizione del primo byte in cui va scritto il valore rilocato.
  - Il *valore* della voce indica l'indirizzo base che, durante la rilocazione, va sommato al valore contenuto nella posizione specificata. Per esempio, può essere l'indirizzo dell'area dati.
- [5:1L] Un file eseguibile è generalmente prodotto dopo una fase di collegamento (*link*) di più file oggetto.
  - Per produrre un eseguibile indipendente dalla posizione si deve partire da oggetti che siano tutti indipendenti dalla posizione.
  - Per produrre un eseguibile rilocabile si deve partire da oggetti rilocabili o indipendenti dalla posizione.
  - Se anche uno solo degli oggetti costituenti ha riferimenti assoluti, allora l'eseguibile prodotto avrà dei riferimenti assoluti.
- Un oggetto non eseguibile ha dei *riferimenti esterni* che devono essere *risolti* per ottenere un eseguibile. Il linker effettua la risoluzione usando la *tabella dei simboli* presente in ogni oggetto. La tabella elenca i riferimenti esterni non risolti e i simboli interni globali di un oggetto.
- Perché da un insieme di oggetti si possa ottenere un eseguibile, ogni riferimento esterno in uno degli oggetti deve corrispondere ad un simbolo interno in qualche altro oggetto. Il linker effettua la risoluzione dei riferimenti esterni e unisce le tabelle di rilocazione degli oggetti in una, se l'eseguibile deve essere rilocabile.
- La tabella dei simboli contiene le voci che descrivono i simboli, che sono le stringhe che identificano una variabile o una funzione. Per ogni voce ci sono dei cammpi che specificano il nome del simbolo, il suo tipo, il valore.
  - Il *nome* del simbolo è una stringa, che è la rappresentazione dell'identificatore del variabile o funzione. Siccome le voci nella tabella dei simboli sono di lunghezza fissa, può darsi che il nome del simbolo sia più lungo del campo. In questo caso il campo non contiene la stringa, ma un puntatore alla stringa. I *simboli lunghi* sono tutti contenuti in un'apposita area.
  - Il *tipo* di voce indica se si tratta di un simbolo globale o indefinito (cioè da risolvere), se si riferisce a dati o a codice.
  - Il *valore* della voce è un indirizzo espresso come offset, cioè una distanza da un punto fisso, per esempio l'inizio del file oggetto. Indica a quale indirizzo corrisponde quel simbolo.
- [6:2L] Un oggetto è formato da diverse sezioni. Le più importanti sono la sezione *text*, la sezione *data* e la sezione *bss*. Ogni informazione di controllo è contenuta in una diversa sezione nel file oggetto. Tutti gli indirizzi in un oggetto rilocabile vengono espressi come offset rispetto all'inizio di una sezione.
- Quando un compilatore produce un oggetto collegabile, questo contiene sia una tabella dei simboli che una tabella di rilocazione. Ogni voce di rilocazione, nel campo di valore, contiene un

simbolo, che è il nome di una sezione o di un simbolo esterno. Il valore è l'indirizzo corrispondente a quel simbolo. Il campo posizione contiene un puntatore all'indirizzo che va rilocato sommandovi il valore del simbolo.

- Struttura di un oggetto collegabile. Struttura di un file eseguibile. Struttura di una libreria.
- [7:2S] Struttura di un file oggetto collegabile e di un file eseguibile in un sistema di tipo Unix. Immagine di un processo in memoria. *Core dump*, file *core*.
- [8:1L] Quando il codice di un programma è troppo grosso rispetto alla memoria disponibile, una tecnica usata spesso nei programmi MSDOS è quella degli *overlay*. Un apposito caricatore legge dal'eseguibile una tabella che elenca quali overlay sono disponibili e li carica in memoria *du domanda*. In pratica, quando viene chiamata una funzione che è definita in un overlay che non si trova in memoria ma su disco, in realtà viene chiamato il caricatore, il quale copia dal disco l'overlay *sovrapponendolo* in memoria ad un altro overlay che non è al momento utilizzato.
- Un concetto analogo è quello, più moderno, delle *librerie dinamiche*, che vengono caricate in memoria su domanda da un apposito caricatore. In generale questa tecnica non è adottata per scarsità di memoria, ma per risparmiare sul tempo di caricamento del programma, cioè per farlo partire più in fretta, e per permettere l'uso di moduli la cui esistenza è nota solo durante l'esecuzione (a *runtime*), e non durante la compilazione (*compile time*). Alcuni linguaggi fanno uso comune di librerie dinamiche, come il *Python* e il *Java*.
- Le *librerie condivise* sono usate in tutti i sistemi moderni di uso generico, come *Unix* e *NT*. Quando il programma è collegato dal linker, l'eseguibile non contiene gli oggetti di libreria, ma solo dei riferimenti simbolici ad essi. Il programma viene caricato da un caricatore rilocante apposito, che traduce i riferimenti alle funzioni di libreria in chiamate a se stesso. Ogni volta che uno di questi riferimenti viene utilizzato, il caricatore verifica che la libreria relativa sia già in memoria, ed eventualmente la carica, e quindi effettua una rilocazione al volo alla libreria, cioè risolve il riferimento all'ultimo momento (*lazy binding*. Il vantaggio è che esiste al più un sola copia in memoria di ogni libreria, con un risparmio di occupazione di memoria, e inoltre i programmi non contengono il codice delle librerie, con un risparmio di occupazione dello spazio disco.
- Perché una libreria condivisa possa essere utilizzata da più programmi, essa deve essere scritta in maniera *rientrante*. Questo significa che tutte le parti condivise non devono essere automodificanti. Mentre questa è una condizione che normalmente è sempre soddisfatta per il codice, non sempre è vero per i dati. Quando si scrive codice rientrante, bisogna mantenere tutti i dati locali sullo stack, oppure allocarli dinamicamente e puntare ad essi con un riferimento che si trova sullo stack o in un registro della CPU.

I processi

S: 4.1…, 4.3…; P: 2.1-2.8, 2.13-2.15

- [9:2L] Programma, (istanze di un) processo.
- Stato di un processo: nuovo, pronto, in esecuzione, bloccato, zombi. Transizioni possibili fra gli stati, blocco, sblocco, prelazione, scheduling.
- Process Control Block: stato della CPU, diritti, allocazioni di memoria, priorità, segnali, file aperti, accounting, blocco su una coda di eventi.
- Creazione di un processo in DOS: interprete dei comandi, TSR. `load_and_exec`, `load_overlay`, `end_prog`, `keep_prog`. Le routine del DOS sono interrompibili ma non rientranti, perché non è pensato per essere multiprogrammato.
- [10:2L] Creazione di un processo in Unix: `fork`, `exec`, `exit`.
- Creazione di un processo in Psos: `t_create`, `t_start`.

Scheduling

S: 4.2…, 21.5…, 22.4, 22.4.1…, 22.5.2; P: 2.9-2.12

- Meccanismo: contesto di un processo, cambio di contesto, dispatcher, dispatch latency.
- Rescheduling scatenato da interrupt: vettore degli interrupt, cambiamento di stato, gestore dell'interrupt, salvataggio del processo in esecuzione, sblocco di un altro processo, rescheduling.
- Politica: algoritmi di scheduling. Time sharing. Round robin, priorità, round robin con priorità. Politiche Posix per le applicazioni real-time: Fifo e Round robin, priorità statiche, `sched_yield`. La politica classica di scheduling in Unix, priorità dinamica, nice level, rescheduling.
- Una generica politica di scheduling: multilevel feedback queue.
- Criteri di valutazione: utilizzo della CPU (efficienza), throughput (es.: numero di transazioni per secondo), tempo di completamento di un task, tempo di risposta.
- [11:1L] Scheduling in Psos: il rescheduling avviene solo in conseguenza di una chiamata di sistema o di un interrupt. Un task può disabilitare la prelazione e la condivisione di tempo. Se la prelazione è abilitata, il task che gira è sempre quello a priorità massima fra i pronti. Si può effettuare round-robin manuale fra task a pari priorità. Un task può essere sospeso. Non è lo stesso che bloccato, perché può essere sospeso anche da un altro task, quindi mentre non è in esecuzione (pronto o bloccato).
- Il kernel esegue una chiamata di sistema per conto del processo. Tutte le azioni che implicano blocco del processo, e conseguentemente rescheduling, devono essere implementate come chiamate di sistema.
- Chiamate non interrompibili (Linux, Psos), interrompibili (Solaris), non rientranti (Dos).
- In Psos, esiste un modulo detto *Dispatcher* che sceglie il primo task dalla coda dei pronti e gli dà il controllo. Ogni chiamata di sistema chiamata da un task esce attraverso il Dispatcher, mentre non esce attraverso il dispatcher se chiamata da un interrupt handler. Un interrupt handler che interrompe un task e che effettua una chiamata di sistema deve uscire attraverso il Dispatcher. Un interrupt handler che interrompe una chiamata di sistema può non uscire attraverso il Dispatcher.
- [14:1S,1E] Studio del manuale Psos: lettura in aula e commento.
  - «The real-time design problem» 2-1, primi cinque paragrafi, e «Multitasking implementation» 2-2, primi due paragrafi.
  - «State transitions» 2-7, tutto, ignorando i riferimenti ad eventi, semafori, memoria.
  - «Dispatch criteria» 2-12, tutto, e «Interrupt exit» 2-58, tutto.
- Esercizio: disegnare il grafo completo degli stati di un task in Psos, considerando i due stati aggiuntivi *Rs* e *Bs*, e descrivere tutte le transizioni che partono ed arrivano a tali stati.
- [16:1S] Compitino sul programma svolto e illustrazione di soluzione.

Sincronizzazione: concetti, mutua esclusione, implementazioni

S: 6.1, 6.2, 6.3

- [12:2L] Spooler di stampa che usa un buffer di nomi di file di lunghezza infinita e due variabili condivise che indicano la base e la cima dello stack dei nomi: *race condition* (*corsa critica*) quando c'è più di uno scrivente. Concetti di *mutua esclusione* e *sezione critica*.
- Condizioni per il funzionamento di un meccanismo di controllo di processi sincronizzati con mutua esclusione, supponendo velocità relative dei processi imprevedibili:
  1. Non più di un processo alla volta dentro la sezione critica.
  2. Un processo fuori della sezione critica non ne può bloccare altri.
  3. Il tempo di attesa per entrare nella sezione critica è finito.
- Il *mascheramento degli interrupt* è un'operazione privilegiata, ed è poco furbo darne la possibilità

ad un processo utente. Inoltre, su sistemi multiprocessore non è generalmente sufficiente. Tuttavia è una tecnica usata dentro il kernel per implementare sezioni critiche dentro gli interrupt handler.

- Due processi che usano una risorsa alternativamente possono usare una variabile di turno (*alternanza*). Questo metodo è utilizzabile anche fra più dispositivi in DMA, per esempio è il metodo usato dal chip Lance per sincronizzare l'accesso ad un anello di buffer. Tuttavia non è generale, perchè un processo non può entrare due volte di seguito nella sezione critica.
- [13:2L] Il caso dello spooler di stampa è del tipo "più produttori, un consumatore". Il caso della CPU e del Lance è del tipo "un produttore, un consumatore in alternanza". Il caso di un processo che riceve richieste remote e poi incarica diversi processi di eseguirle è del tipo "un produttore, più consumatori". Il caso di diversi processi che accedono ad una scheda sonora non è di nessuno di questi tipi: i problemi di sincronizzazione sono molto vari.
- Soluzione particolare: produttore-consumatore con un anello di lunghezza finita e due variabili `in` e `out` che sono indici nell'array. Si spreca un posto, ma funziona per un singolo produttore ed un singolo consumatore.

```
data buffer[N];
int in = 0;
int out = 0;

insert(data) {                        // codice usato dal produttore
  while ((in+1)%N == out) continue;   // enter critical section
  buffer[in]=data;                    // sezione critica
  in=(in+1)%N;                        // exit critical section
}

data extract() {                      // codice usato dal consumatore
  while (out == in) continue;         // enter critical section
  retval=buffer[out];                 // sezione critica
  out=(out+1)%N;                      // exit critical section
  return retval;
}
```

- Un *mutex* è un criterio (struttura dati più algoritmo) che permette l'accesso in mutua esclusione ad una sezione critica. Proviamo ad implementarlo con una *variabile di lock*:

```
int lock = 1;
…
while (lock == 0) continue; lock = 0; // enter critical section
…                                      // sezione critica
lock = 1;                             // exit critical section
```

Non funziona, perché ci riporta esattamente allo stesso problema che si tenta di risolvere. Se però si rende atomica l'operazione sul lock, allora funziona (su monoprocessore):

```
int lock = 1;
…
while ((lock -= 1) < 0) lock += 1;    // enter critical section
…                                      // sezione critica
lock += 1;                            // exit critical section
```

Tuttavia, il tempo di attesa non è limitato, quindi contravviene alla regola numero 3.

- [15:1L,1E] In alternativa, su una macchina monoprocessore, basta disabilitare gli interrupt per rendere atomiche le operazioni sulla variabile di lock. Su multiprocessore non è pratico, e quindi è necessario supporto hardware: test and set lock (TSL) per risolvere la concorrenza sullo stesso processore e su diversi processori.
- Nota fuori dal programma del corso: esiste una soluzione generale per la mutua esclusione fra due processi. Consiste nel combinare la variabile di turno con le variabili booleane di prenotazione. È una soluzione che soddisfa tutte le tre condizioni dette:

```
// Ci sono 2 processi: un processo si chiama 0 e l'altro si chiama 1.
// Quindi se uno si chiama p, l'altro si chiama 1-p.
int turn = 0;
boolean interested[2] = { false, false };

void enter_critical_section (int p)
{
  interested[p] = true;
  turn = p;
  while (interested[1-p] && turn == p)
    continue;
}

void exit_critical_section (int p)
{
  interested[p] = false;
}
```

- Esiste anche un algoritmo, detto *the bakery algorithm* (algoritmo del fornaio) che soddisfa tutte e tre le condizioni dette per un numero arbitrario di processi.

Sincronizzazione: semafori e stallo (deadlock)

S: 6.4…, 6.5.1, 6.5.2, 6.7(l'inizio), 7.4.2, 7.4.4; P: 2.49-2.50

- Definizione di semaforo implementato con attese attive:

```
int sem;

P(sem) {              // wait
  while (sem <= 0) continue;
  sem -= 1;
}

V(sem) {              // signal
  sem += 1;
}
```

- Un semaforo implementato senza attese attive. Sleep e wakeup.

```
struct {
  int val;
  queue q;            // coda di processi in attesa
} sem;

P(sem) {              // wait
  sem.val -= 1;
  if (sem.val < 0) {
    enqueue(sem.q);   // accoda questo processo
    sleep();          // bloccalo
  }
}

V(sem) {              // signal
  sem.val += 1;
  if (sem.val <= 0) {
    p = dequeue(sem.q); // scoda uno dei processi in coda
    wakeup(p);          // sbloccalo
  }
}
```

- Questa a prima vista non è una soluzione, nel senso che abbiamo solo spostato il problema, in

quanto c'è comunque bisogno di un mutex sulla variabile `sem`. Infatti, in entrambe le soluzioni proposte, le istruzioni di test e di decremento (o incremento) sono una sezione critica, da eseguire in mutua esclusione.

- Per garantire la mutua esclusione, sarà necessario disabilitare gli interrupt su una macchina monoprocessore, o usare un vero e proprio mutex usando l'istruzione TSL su una macchina multiprocessore. Tuttavia questa sezione critica è piccolissima, e l'uso di attese attive o mascheramento è perfettamente tollerabile.
- Esercizio: si può implementare un mutex con un semaforo inizializzato ad 1.
- Esercizio: si può ottenere sincronizzazione fra due lavori che devono essere eseguiti in una data sequenza (ordinamento) usando un semaforo inizializzato a 0.
- Esercizio: si può implementare un *rendevous* fra due task usando due semafori inizializzati a 0. Attenzione: sbagliando l'ordine delle operazioni si può causare *stallo* (*deadlock*) deterministico o no.
- [17:2E] I semafori usati per risolvere i problemi visti finora assumono due soli valori: 1 e 0. Semafori di questo tipo vengono detti *semafori binari* (counting semaphores). Semafori che invece assumono anche valori maggiori di 1 sono detti *semafori a contatore* (counting semaphores).
- Esercizio: un produttore, un consumatore, buffer di lunghezza *N* usato senza sprechi: si usano due variabili indice e due semafori, uno di pieni inizializzato a 0 e uno di vuoti inizializzato a *N*.
- Esercizio: più produttori e consumatori, buffer di lunghezza *N*. Non si possono usare le soluzioni di prima, perché c'è conflitto sugli indici. Una soluzione semplice è mettere un mutex sugli indici. Una soluzione che consenta l'esecuzione di un produttore ed un consumatore contemporaneamente può esser fatta con due mutex, uno sull'indice di inserzione e uno su quello di estrazione.
- [18:1L] Implementazione dei semafori in Psos: semplice incremento o decremento. Code FIFO o a priorità. Operazione bloccante, non bloccante, bloccante con timeout.
- Ecco come potrebbe essere implementato un semaforo dentro il kernel del sistema operativo:

```
P(sem) {                              // funzione wait
  loop {                              // ciclo infinito
    mask_interrupts();                // inizio sezione critica
    if (sem.count > 0)                // test del contatore
      sem.count -= 1;                 // decremento
    else                              // semaforo bloccante
      this_process.state = blocked;   // cambia lo stato del processo
    unmask_interrupts();              // fine sezione critica
    if (this_process.state == running)// se il processo non ha cambiato stato
      return;                         // torna al processo chiamante
    enqueue(this_process,sem.queue);  // mette questo processo in coda
    dispatch();                       // sceglie il prossimo processo
  }                                   // quando riprende il controllo riprova
}

V(sem) {                              // funzione signal
  mask_interrupts();                  // inizio sezione critica
  sem.count += 1;                     // incrementa il contatore
  unmask_interrupts();                // fine sezione critica
  if (sem.queue == NULL)              // se non c'è nessuno in coda
    return;                           // torna al processo chiamante
  this_process.state = ready;         // cambia lo stato del processo
  enqueue(this_process,ready.queue);  // e lo mette nella coda dei pronti
  p = dequeue(sem.queue);             // scoda il primo processo bloccato
  p.state = ready;                    // e ne cambia lo stato
  dispatch();                         // sceglie il prossimo processo
}
```

- Nell'esempio precedente, mascherare gli interrupt è la cosa più pratica in un sistema monoprocessore. Se ci sono più processori che usano i semafori, è conveniente usare altri metodi, come un ciclo attivo su una variabile acceduta con l'istruzione TSL.
- [19:1E,1L] Come si implementa un driver di orologio in Psos usando un semaforo, un interrupt

handler e un task, senza protezione del kernel: il task crea il semaforo con contatore pari a 0 e inizializza l'hardware. Quindi fa una P sul semaforo, bloccandosi. Quando avviene un interrupt, l'interrupt handler fa una V sul semaforo e sblocca il task, il quale fa il proprio lavoro e si riblocca, e così via.

- Due processi che devono entrambi bloccare le stesse due sezioni critiche, che devono essere accessibili anche da altri: possibile stallo. Si evita impegnando le sezioni critiche sempre nello stesso ordine, ma in situazioni complesse può essere impossibile.
- In generale, se N risorse sono condivise fra più processi che ne impegnino più di una alla volta, esiste il rischio di stallo. È possibile eliminare la possibilità di stallo definendo un *ordinamento totale* fra le risorse, cioè numerandole da 1 a N, ed imponendo che ogni processo che impegni più di una risorsa lo faccia in ordine numerico, dalla risorsa col numero più basso al più alto. Tuttavia non è sempre possibile scrivere i processi in modo che rispettino questo vincolo.
- Un altro modo di evitare lo stallo è di imporre che tutti i processi che impegnano più di una risorsa utilizzino il *two phase lock protocol*, cioè non si blocchino mai tentando di impegnare una risorsa se ne hanno già impegnate delle altre: o impegnano tutte quelle di cui hanno bisogno senza bloccarsi, o non ne impegnano nessuna. Un modo di implementare il two phase lock protocol consiste nell'usare i semafori in modalità non bloccante e usare un ciclo attivo.
- Esercizio: lettori e scrittori, precedenza ai lettori.

```
m = 1;                  // inizializza per uso come mutex
scrittura = 1;          // inizializza per uso come mutex
lettori = 0;            // numero dei lettori attivi

scrivi() {              // routine di scrittura
  P(scrittura);         // inizio sezione critica di scrittura
    …                  // effettua operazione di scrittura
  V(scrittura);         // fine sezione critica di scrittura
}

leggi() {
  P(m);                 // inizio sezione critica accesso a lettori
  lettori += 1;         // si registra come nuovo lettore
  if (lettori == 1)     // se non c'erano già altri lettori
    P(scrittura);       // blocca l'accesso agli scrittori, o aspetta
  V(m);                 // fine sezione critica accesso a lettori
    …                  // effettua operazione di lettura
  P(m);                 // inizio sezione critica accesso a lettori
  lettori -= 1;         // registra l'uscita come lettore
  if (lettori == 0)     // se non c'è più nessun lettore
    V(scrittura);       // sblocca l'accesso agli scrittori
  V(m);                 // fine sezione critica accesso a lettori
}
```

- Implementazione dei semafori in IPC (Posix). È possibile eseguire atomicamente più operazioni su un vettore di semafori, semplificando così la soluzione di alcuni problemi di stallo. Non c'è garanzia sull'ordine in cui i processi vengono rilasciati. Operazione bloccante o no, a scelta. Incremento o decremento di quantità arbitrarie con l'uso diuna singola operazione che si chiama `semop`. L'uso classico è ottenuto incrementando di 1 per la V e decrementando di 1 per la P. Usando argomento 0 si ottiene un funzionamento come variabile di lock.
- Nota fuori dal programma del corso: un monitor è un costrutto che permette la mutua esclusione su una sezione critica e operazioni di wait dal sistema operativo, o anche solo dal linguaggio, ed è un'alternativa più pulita ed a livello più alto rispetto ai semafori. Tuttavia è poco usata.

Sincronizzazione e comunicazione: messaggi

S: 4.6.1, 4.6.2, 4.6.3, 4.6.4; P: 2.44;2.46

- [20:1L,1E] Messaggi come primitive di sincronizzazione: semplice ordinamento parziale fra due processi senza mailbox, mutua esclusione con mailbox condivisa, accesso multiplo ad una sezione critica con mailbox condivisa. Si può dimostrare (noi non lo facciamo) l'equivalenza coi semafori ed i monitor.
- Esercizio: implementare il caso un produttore - un consumatore con buffer finito e messaggi vuoti. Mettendo dentro i messaggi l'indice del posto nel buffer si può avere concorrenza totale fra più produttori e più consumatori.
- Un sistema a scambio di messaggi ha due aspetti principali: la capacità di *sincronizzazione* e la capacità di *scambio di informazione*. Ogni sistema può enfatizzare l'una o l'altra a seconda dello scopo per cui è pensato o ottimizzato. Qui sono elencate diversi modi in cui si distinguono i sistemi a scambio di messaggi. Normalmente, ogni sistema implementa solo alcune delle numerose possibilità.
  - Può esistere la possibilità di creare esplicitamente una mailbox globale, cioè accessibile da più processi. Oppure una per processo implicita, cioè creata automaticamente dal sistema.
  - Può esistere la possibilità di scegliere di ricevere un messaggio da chiunque, o da un mittente specifico, o da un gruppo di mittenti.
  - Ogni sistema ha una sua convenzione per scegliere i nomi dei corrispondenti e della casella postale. In Psos si può associare un nome ad ogni mailbox, e si può chiedere qual è l'identificatore di mailbox associato ad un dato nome. Su Internet ogni nodo ha un indirizzo.
  - Le mailbox possono avere diritti. Ad esempio, il diritto in scrittura su una mailbox permette ad un processo di inviare un messaggio, un diritto in scrittura permette di riceverlo.
  - Le mailbox possono avere dimensione infinita, cioè limitata solo dalla memoria totale nel sistema, o finita, cioè contenere un numero massimo di messaggi, o una dimensione complessiva massima dei messaggi in coda. Nel caso di mailbox infinita, solo la ricezione è bloccante, e la trasmissione restituisce un errore indicante che il sistema ha esaurito la memoria, se questo è il caso. Nel caso di mailbox finita, generalmente anche la trasmissione è bloccante.
  - Nel caso più generale, il sistema operativo copia il messaggio dallo spazio di memoria del mittente nella mailbox all'atto dell'invio, e dalla mailbox allo spazio di memoria del destinatario all'atto della ricezione. Effettuare due copie da memoria a memoria per ogni messaggio è poco efficiente, specie per messaggi lunghi o in casi nei quali i messaggi siano numerosi. Esistono quindi casi particolari in cui il sistema operativo cerca di risparmiare su queste operazioni.
  - Uno di questi casi è quello del *rendezvous*, che si implementa con una mailbox finita di lunghezza nulla. L'effetto è che la trasmissione è bloccante se non c'è nessun ricevente già bloccato, e viceversa. Siccome mittente e destinatario completano ognuno la propria oeprazione contemporaneamente, non è mai necessario effettuare una doppia copia, né d'altra parte sarebbe possibile, essendo la mailbox di lunghezza nulla. Il sistema può effetutare una singola copia dallo spazio di memoria del mittente a quello del destinatario, o mappare lo spazio del mittente in quello del destinatario, risparmiando del tutto la copia.
  - La dimensione dei messaggi può essere fissa nel sistema, fissa per ogni mailbox, o variabile da messaggio a messaggio.
  - Le tipiche condizioni di errore di trasmissione o ricezione sono: mailbox distrutta, interruzione della chiamata, mancanza di buffer di sistema
  - Nel caso di sistemi orientati più alla comunicazione che alla sincronizzazione, e specie nel caso di reti di comunicazioni, si distingue fre comunicazione con connessione (*connection oriented*) oppure a singoli messaggi (*connectionless*). Nel primo caso i corrispondenti aprono una connessione, partecipando entrambi a questa operazione, dopodichè si scambiano messaggi su quella connessione, senza dover ulteriormente specificare il

destinatario. Nel secondo caso, ogni messaggio ha storia propria, e deve contenere l'indirizzo del estinatario.
    - Sempre nel caso di reti, si distingue fra connessioni affidabili e non. In una comunicazione affidabile, il sistema di scambio di messaggi si preoccupa di verificare la ricezione del messaggio. Se questo non arriva a destinazione, ci riprova e alla fine segnala un errore al mittente.
- [21:1L] Implementazione dei messaggi in Psos: code a lunghezza limitata per coda o per sistema, messaggi senza tipo lunghi 16 byte, code dei task gestite con politica FIFO o secondo la priorità dei task. La send non è mai bloccante, la receive può essere bloccante, non bloccante, bloccante con timeout. Il messaggio può esser messo in testa o in coda alla coda di messaggi. Si può mandare un broadcast di un messaggio a tutti i task in attesa su una coda: se non ce ne sono, il messaggio è perduto, altrimenti tutti i task ricevono una copia del messaggio, e la coda rimane vuota e senza task in attesa.
- Implementazione dei messaggi in IPC (Posix): messaggi di lunghezza arbitraria, code di lunghezza limitata, messaggi con tipo, ricezione su qualunque messaggio, solo tipo dato, tutti eccetto tipo dato, tipo pari ad almeno il dato. Operazioni bloccanti o no, a scelta. A scelta, il messaggio da ricevere troppo lungo dà errore o tronca il messaggio.

Sincronizzazione: inversione di priorità

S: 5.5

- Un processo ad alta priorità *A* è bloccato all'ingresso di una sezione critica occupata da un processo a bassa priorità *B*. Quest'ultimo, dentro la sezione critica, subisce prelazione da parte di un processo a media priorità *M*. L'effetto è che *M* si comporta come se avesse priorità maggiore di *A*, avviene cioè una *inversione di priorità*. Alcune cure possibili:
    - Ogni processo disabilita la prelazione prima di entrare nella sezione critica, e la ripristina all'uscita.
    - Ogni processo alza la propria priorità ad un valore *p* prima di entrare nella sezione critica, e la rispristina all'uscita, dove *p* è un valore pari alla massima fra le priorità dei processi che utilizzano la sezione critica.
    - Se esiste una primitiva di sincronizzazione che controlla la sezione critica nel sistema operativo, nel linguaggio (per esempio con un costrutto monitor) o in una libreria, una delle due operazioni sopradette può essere eseguita automaticamente.
    - Sempre se la sezione critica è una primitiva, si può applicare il *priority inheritance protocol*, cioè quando *A* si blocca sulla sezione critica, *B* eredita la priorità di *A* fino alla sua uscita dala sezione critica.

Sincronizzazione: esercizi

- [22:2E] Domande con risposta abbastanza definita, del tipo di quelle dei compitini:
    - Cos'è una sezione critica?
    - Cos'è un mutex? Farne un esempio di implementazione.
    - Cos'è un semaforo?
    - Quali valori può assumere il contatore di un semaforo?
    - Cos'è un rendezvous?
    - Che differenza c'è fra un rendezvous implementato con una mailbox di dimensione massima nulla in Posix e in Psos?
- Domande aperte, o con il trucco, o che necessitano di una discussione non brevissima:
    - Un processo in una sezione critica è interrompibile?
    - Un flusso di esecuzione a basso livello (nel kernel o in un driver) che si trova in una sezione

critica è interrompibile da un interrupt?
    - In Psos, un interrupt handler può fare chiamate di sistema?
- [23:1S] Compitino sul programma svolto e illustrazione di soluzione.

Panoramica sulla gestione della memoria

S: 8.2, 8.3, 8.4…, 8.5.1, 8.5.2…, 9.1, 9.4, 9.5.3, 9.7, 9.7.1, 9.8.1

- [24:2L] Due modi principali di tener conto delle aree di memoria occupate: ogni area contiene un puntatore all'area successiva, oppure con tabella di allocazione. La tabella contiene puntatore e lunghezza, o solo puntatore, o solo lunghezza (*partizioni variabili*). Oppure la memoria è allocata in blocchi(*partizioni fisse*), e la tabella registra se il blocco è libero o occupato. Uso della memoria in DOS, lista dei processi in memoria e delle aree di memoria libere e occupate (PSP).
- L'uso di partizioni fisse causa *frammentazione interna*, l'allocazione senza *compatazione* cause *frammentazione esterna*. Criteri di allocazione: first fit, next fit, best fit, worst fit.
- Swap dell'immagine di un processo su disco. Paginazione: scompare la frammentazione esterna, si può fare swap di parte di un processo. Esempi di memoria virtuale più grande o più piccola della memoria fisica, tabella delle pagine come lista di coppie fisico-virtuale, bit di validità, bit di presenza. Pagina mancante e sua gestione, massimo numero di page fault generati da una singola istruzione.
- [25:2L] Località dei riferimenti, working set, thrashing. Algoritmo LRU, bit di riferimento, invecchiamento come approssimazione di LRU. Prepaginazione, paginazione su domanda, bit di modifica. Il codice di un programma può non essere mai messo in swap, ma restare sul file system.

Memoria virtuale: paginazione

S: 8.5.5, 9.2, 9.3, 9.8.5, 9.8.6

- Meccanismo: le voci della tabella delle pagine non contengono l'indirizzo virtuale da tradurre, che invece è usato come indice. Dimensione eccessiva, accenno alle pagine a più livelli, alla cache della MMU e alla possibilità di swap delle tabelle stesse. Bit di cache abilitata, pagina valida, riferita, modificata, presente, diritti (o protezione).
- [26:1L] Tabella delle pagine per processo ed inverted page table per sistema (tabella dei frame) dove fra l'altro c'è scritto per ogni frame se è libero oppure a quale processo, dispositivo o altro è attualmente allocato.
- Memoria virtuale per il real-time: costo della traduzione fisico↔virtuale effettuata dall'MMU, della necessità di traduzioni dentro il kernel per parlare coi task e coi dispositivi di i/o, maggior costo del cambio contesto. Protezione preferibile in ambienti complessi o eterogenei.
- Un *bus error* è generato per accesso a memoria fisica o virtuale inesistente. Protezione fra task: spazi separati; dentro il task: indirizzo `0` non valido e sola lettura per il codice. *Segmentation violation* ver violazione dei diritti.
- Cambio di contesto implica svuotamento della cache di memoria. Con memoria virtuale implica anche svuotamento della cache di MMU. Con swapping può anche causare swap-in delle pagine di memoria del nuovo processo ed anche delle sue tabelle delle pagine.
- Swap device o backing store. Swap su più partizioni. Memoria virtuale coincidente con lo spazio di swap (BSD) o memoria fisica più swap (Linux). I dati non variabili (codice) possono essere lasciati sul disco al loro posto o subire swap-out una volta in memoria per velocizzarne l'accesso (BSD). Blocco delle pagine in memoria su chiamata di sistema, buona per real-time e per DMA su aree di utente. Mai usare swap per real-time.
- Copia su scrittura ottimizza l'uso della memoria e minimizza le copie inutili, massimizzando la

condivisione delle pagine. Su fork, il nuovo processo ha tutte le pagine in sola lettura. Ad ogni segmentation violation, si alloca una nuova pagina che è una copia della vecchia, altrimenti la pagina resta condivisa. Se c'è copia su scrittura, è importante separare i dati in sola lettura dagli altri.

- [27:1L,1E] Esercizio: in quali casi è generata un'eccezione di pagina mancante?
- Esercizio: mentre l'accesso ad una pagina non valida genera un'eccezione di errore di bus (*bus error*), l'accesso ad una pagina non presente genera un'eccezione di pagina mancante (*page fault*), che si usa per gestire la paginazione su domanda. Descrivere più edttagliatamente possibile la sequenza di azioni che porta alla sostituzione di una pagina dallo swap.
  1. Accesso a pagina *valida* ma non *presente*, con *diritti* corretti.
  2. La MMU genera un *page fault*.
  3. La CPU gestisce l'eccezione e chiama il gestore.
  4. Il gestore verifica che ci sia un frame libero.
     a) Se non c'è un frame libero, si decide quale pagina va messa sullo swap per liberare un frame.
        - Se la pagina da liberare è sporca (modificata), si inizia il suo trasferimento sullo swap.
        - Cambio di contesto, la CPU va ad un altro processo. Il primo processo è bloccato in attesa non interrompibile.
        - Arriva un interrupt che segnala la fine del trasferimento della pagina sporca sul disco.

     b) Che sia avvenuto lo swap o no, il frame da liberare è marcato come libero, e la pagina a cui il frame era associato è marcata come non presente.
  5. La pagina che ha generato il page fault poteva essere:
     - di memoria non inizializzata mai acceduta prima: la si azzera;
     - di stack mai usato prima: non si fa niente;
     - di codice o dati in sola lettura o di dati mai sporcati: la si legge dal file system (dal disco);
     - di dati sporcati: la si legge dallo swap (dal disco).

     a) Se è sul disco, si inizia il trasferimento da disco.
     b) Cambio di contesto, la CPU va ad un altro processo. Il primo processo è bloccato in attesa non interrompibile.
     c) Arriva un interrupt che segnala la fine del trasferimento da disco.
  6. La pagina viene marcata come valida e non modificata, e associata al frame che era stato liberato, e che ora è marcato non libero.
  7. Il processo che aveva generato il page fault è sbloccato e messo nella coda dei pronti.
- Esercizio: in quali casi è generata un'eccezione di errore di bus?
- Esercizio: in quali casi è generata un'eccezione di violazione di segmentazione?
- Esercizio: come si implementa la memoria condivisa fra processi?

Chiamate di sistema e di libreria per gestire la memoria

S: 8.6…, 8.7, 9.5.4…, 9.6, 9.6.1; P: 2.39, 2.42-2.43

- [28:2L] Le chiamate per la creazione, aggancio e sgancio di memoria condivisa in IPC: `shmget`, `shmat` e `shmdt`.
- Allocazione di memoria in Unix: la chiamata di sistema `brk`. Le funzioni Posix per l'allocazione della memoria: `malloc`, `free`, `calloc`, `realloc`.
- Allocazione di memoria in Psos. Regioni contigue di memoria fisica, segmenti allocabili di dimensione arbitraria con attesa FIFO o a priorità: `rg_create`, `rg_delete`, `rg_getseg`, `rg_retseg`. Partizioni contigue di memoria fisica, buffer allocabili di dimensione fissa senza attesa: `pt_create`, `pt_delete`, `pt_getbuf`, `pt_retbuf`.

Device

S: 12.3… P: 4.1-4.2, 4.5-4.8, 4.12

- Device a blocco, da cui ci si aspetta `read`, `write`, `seek`; a carattere, da cui ci si aspetta `get`, `put`.
- In Posix e in Psos, un driver è implementato mediante le interfacce di `open`, `close`, `read`, `write` e `ioctl`, più una routine di gestione dell'interrupt.
- Per scrivere un driver in Posix e Psos basta scrivere le cinque routine di `open`, `close`, `read`, `write`, `ioctl` e l'interrupt handler. Il sistema operativo si occupa dell'interfaccia. La `open` e la `close`, se non servono, possono essere vuote. Per semplici dispositivi implementano l'accesso serializzato alla risorsa, o effettuano controlli di diritti prima di garantire l'accesso.
- [29:1L] Come abbiamo visto, le interfacce dei driver sono uniformi. La cosa interessante è che l'interfaccia è come quella usata per l'accesso al file system. Grazie a questa caratteristica è possibile implementare la *device independence*. In Unix è implementato anche il concetto di *uniform naming*.
- Oltre al tipo di accesso ai device visto, ce ne sono altri più moderni di uso specifico, come i device mappati in memoria, ad esempio in Unix con `mmap`, `munmap`. Un'altro tipo di interfaccia molto diffusa sono i *socket*, che sono usati soprattutto per le comunicazioni in rete.
- Operazioni su device: *bloccanti*; *nonbloccanti sincrone*, usate per il polling; *nonbloccanti asincrone*. Una semantica si dice bloccante quando il task si blocca sull'operazione e quando si sblocca (senza errore) i dati sono stati scritti o letti dal buffer. Si dice non bloccante sincrona quando l'operazione non è bloccante e torna con una errore, indicando che nessuna azione è stata completata, oppure senza errore, indicando che l'azione è stata completata. Permette di implementare politiche di polling. La semantica non bloccante asincrona è non bloccante e torna normalmente senza errore, ma l'azione è completata in un tempo successivo. Il driver fornisce al task informazioni sui comandi completati usando qualche meccanismo, per esempio un comando di `ioctl`, o una coda di messaggi, oppure un *callback*, cioè un meccanismo che consente al driver di chiamare una funzione del task. La `select` di Posix consente il blocco contemporaneo su più device, anche con timeout.

I file system

S: 10.1.1, 10.1.2, 11.2.2, 11.2.3, 10.4.2, 10.4.4, 13.5

- [30:2L] I file come interfaccia uniforme ai dispositivi del kernel: *file speciali* (device) e *pseudo device*, *pipe*.
- Le funzioni di filesystem vedono sotto di loro il driver, che presenta il disco come una sequenza lineare di settori, e presentano sopra un'interfaccia con `open`, `close`, `read`, `write`, `seek`, `lock`.
- Implementazione dei file system: tabella di allocazione dei file. I *cluster* (o *settori logici*) sono composti da uno o generalmente più *settori* (o *settori fisici*). La *FAT* in DOS: 12, 16, 32 bit. Voci da 16 bit con cluster da 32 KB ⇒ partizione da 2GB al più.
- In DOS si alloca il file con algoritmo first fit. In ext2fs si usa worst fit.
- Le directory in DOS, accenno alle directory in Unix.
- Accesso *sequenziale* e *casuale*. Tabella dei file aperti di sistema, con le caratteristiche del file. Tabella dei file per processo, con puntatore per accesso sequenziale. Accenno ai problemi per l'accesso concorrente.
- Diritti: lettura, scrittura, esecuzione. Diritti di uso dei file in Unix e in NT (liste di accesso).
- Backup, restore: backup *completo*, *differenziale*, *incrementale*.

Laboratorio: analisi di un device driver per il chip LANCE

- [31:1E,1S] Cosa serve per implementare un driver per il LANCE: open, close, interrupt handler, read e write sincrone e asincrone.
- Operazioni di i/o sincrone e asincrone. Code e semafori in Psos, operazione t_mode usata per disabilitare la prelazione.

Esercizio: qual è la sequenza di eventi che ha luogo quando si alloca memoria con una chiamata di realloc?

Esercizio: qual è la sequenza di eventi che ha luogo quando si apre un file in un sistema operativo di tipo Unix? E quando si fa una read?

1. Si guarda se nel sistema quel file è stato già aperto.
   a) Se non è stato già aperto, il sottosistema che gestisce il file system nel S.O. cerca nella directory root, che è in una posizione fissa sul disco, la voce corrispondente alla prima directory del percorso. Individua la posizione del primo cluster e dei successivi, guardando nella FAT. Poi cerca nella prima directory la voce corrispondente alla seconda, e così via fino a trovare il file cercato.
   b) Crea una voce nella tabella di sistema dei file aperti, e vi memorizza le caratteristiche del file, in maniera da non doverle più guardare nelle directory e nella FAT.
2. Crea una voce nella tabella dei file aperti del processo che punta alla voce corrispondente al file nella tabella globale, e inizializza il puntatore di lettura e scrittura a 0, cioè all'inizio del file.
3. Restituisce al processo chiamante un identificatore della voce nella tabella del processo dei file aperti.
4. Quando si fa una read, il sottosistema di file system guarda il puntatore di lettura e scrittura del file utilizzato, poi guarda i dati contenuti nella tabella globale dei file aperti, e usa queste informazioni per determinare quali settori del disco bisogna leggere.
5. Poi chiede al sottosistema di i/o di iniziare la lettura dal disco. Se i settori necessari sono stati letti recentemente, potrebbero essere già nella cache del disco.
6. Altrimenti si danno le istruzioni necessarie al controller e si aspetta un interrupt, bloccando il processo e cambiando contesto. Quando arriva l'interrupt, i dati letti vengono messi nella cache del disco.
7. Il S.O. legge i dati dalla cache del disco e li copia nello spazio di memoria del processo.

Esercizio: apro un file in lettura e scrittura e ottengo un errore perché non ho diritti di scrittura. Cosa è successo esattamente?

[32:1E,1S] Compitino sul programma svolto e illustrazione di soluzione. Panoramica riassuntiva del corso e chiarimenti sui punti oscuri.