ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

Διαδιεργασιακή Επικοινωνία

Υλικό από:
Tanenbaum, *Modern Operating Systems,Structured Computer Organization*
Stallings, *Operating Systems: Internals and Design Principles*.
Silberschatz, Galvin and Gange, *Operating Systems Concepts.*
Deitel, Deitel and Choffnes, Operating Systems
Λειτουργικά Συστήματα, Γ.Α. Παπαδόπουλος, Πανεπιστήμιο Κύπρου
Λειτουργικά Συστήματα, Κ. Διαμαντάρας, ΤΕΙΘ
Systems Programming in C, A.D. Marshal, University of Cardiff

Σύνθεση
Κ.Γ. Μαργαρίτης, Τμήμα Εφαρμοσμένης Πληροφορικής, Πανεπιστήμιο Μακεδονίας

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Διαδιεργασιακή επικοινωνία (IPC) (1)

Οι διεργασίες (και τα νήματα) κατ' αρχήν εκτελούνται ανεξάρτητα και δεν μπορούν να συνεργαστούν παρά μόνο μέσω υπηρεσιών του λ.σ. Η συνεργασία είναι απαραίτητη για το συντονισμό των δραστηριοτήτων του συστήματος ή της εφαρμογής.

Η συνεργασία διεργασιών λέγεται Διαδιεργασιακή Επικοινωνία (Inter Process Communication, IPC)

Η IPC είναι απαραίτητη σε λ.σ. πολυ- προγραμματισμού καθώς και σε παράλληλα κατανεμημένα λ.σ.

Η διαδιεργασιακή επικοινωνία θέτει δύο ειδών ζητήματα:

(α) Ποιοί μηχανισμοί πρέπει να είναι διαθέσιμοι στο λ.σ.;

(β) Ποιές κλήσεις συστήματος πρέπει να είναι διαθέσιμες στο API;

Τα API's για τη δημιουργία διεργασιών και για τη διεργασιακή επικοινωνία αποτελούν τη βάση του συντρέχοντος (concurrent) προγραμματισμού.

Διαδιεργασιακή επικοινωνία (IPC) (2)

Η διαδιεργασιακή επικοινωνία παρέχει ένα σύνολο μηχανισμών που επιτρέπει στις διεργασίες

- να συγχρονίζουν την εκτέλεσή τους
- να μεταβιβάζουν δεδομένα
- να διαμοιράζονται δεδομένα
- να διαμοιράζονται πόρους

Η διαδιεργασιακή επικοινωνία προσφέρει πλεονεκτήματα όπως

- αρθρωτή ανάπτυξη κώδικα
- ταχύτερη εκτέλεση εφαρμογών
- πλαίσιο επίλυσης προβλημάτων πολυπρογραμματισμού και παράλληλων ή κατανεμημένων λ.σ.

Σήματα (signals) (1)

Τα σήματα (signals) είναι διακοπές λογισμικού (software interrupts). Τα σήματα μπορεί να είνα σύγχρονα, δηλαδή να προκαλούν συγχρονισμό διεργασιών, αλλά συνήθως είναι ασύγχρονα, δηλαδή οι διεργασίες που τα παραλαμβάνουν τα χειρίζονται στο δικό τους χρόνο.

Τα σήματα έχουν διάφορα επίπεδα προτεραιότητας. Για παράδειγμα, σήματα που παράγονται όταν το σύστημα διαγνώσει κάποιο σφάλμα (bus error, memory fault, illegal operation, ..) ή όταν ο χρήστης ζητήσει αναστολή ή τερματισμό διαργασίας (ctrl-c ή kill στο Unix) θεωρούνται επείγοντα και η παραλαμβάνουσα διεργασία πρέπει τα εξυπηρετήσει άμεσα.

Το σύστημα ορίζει τον αριθμό, τον τύπο και τη προτεραιότητα των σημάτων. Ορισμένα σήματα με σχετικά χαμηλό επίπεδο προτεραιότητας μπορούν να αποκλειστούν, να αγνοηθούν ή να καθυστερήσει η εξυπηρέτησή τους, με ευθύνη του προγραμματιστή (maskable -nonmaskable signals/interrupts). Τα σήματα που αποκλείονται ορίζονται σε μιά 'μάσκα' που τη διαχειριζόμαστε με ειδικές συναρτήσεις. Σε περίπτωση που η διεργασία παραλαβής δεν προβλέπει κάποιο χειρισμό ενός σήματος, το πιθανότερο είναι οτι η παραλαβή του σήματος καταλήγει στο τερματισμό της διεργασίας.

Σήματα (signals) (2)

Γενικά το σύστημα έχει μια προεπιλεγμένη ενέργεια που ακολουθεί τη παραλαβή κάθε σήματος:

Το σήμα απλά διαγράφεται
Η διεργασία τερματίζεται
Δημιουργείται ένα αντίγραφο του χώρου εργασίας της διεργασίας (core file στο Unix) και η διεργασία τερματίζεται
Η διεργασία αναστέλεται

Τα σήματα χωρίζονται σε πέντε βασικές κατηγορίες:

Συμβάντα Υλικού
Συμβάντα Λογισμικού
Συμβάντα Εισόδου/Εξόδου
Διαχείριση Διεργασιών
Διαχείριση Πόρων

POSIX signals (1)

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

SIGHUP 1 **SIGINT 2** **SIGQUIT 3** **SIGILL 4** **SIGABRT 6**

SIGKILL 9 **SIGALRM 14** **SIGCONT 19** **SIGGCHLD 20**

POSIX signals (2)

Δύο βασικές κλήσεις συστήματος διαχείρισης σημάτων:

Αποστολή: kill(pid, sig)

Κλήση συστήματος που στέλνει το σήμα sig στη διεργασία pid. Αν pid > 0 το σήμα αποστέλεται στην αντίστοιχη διεργασία. Αν το pid = 0 το σήμα αποστέλεται σε όλες τις διεργασίες, εκτός από αυτές του συστήματος.

Παραλαβή: sigaction(sig, &action)

Κλήση συστήματος που παραλαμβάνει το σήμα sig και το διαχειρίζεται με βάση τη function(). Υπάρχουν και προκαθορισμένες συναρτήσεις..

Κλήσεις συστήματος διαχείρισης διεργασιών και σημάτων POSIX

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

s : error code

pid : process ID

residual χρόνος που απομένει

Συνεργασία και ανταγωνισμός

Η διαδιεργασιακή επικοινωνία μπορεί να εκφραστεί με:

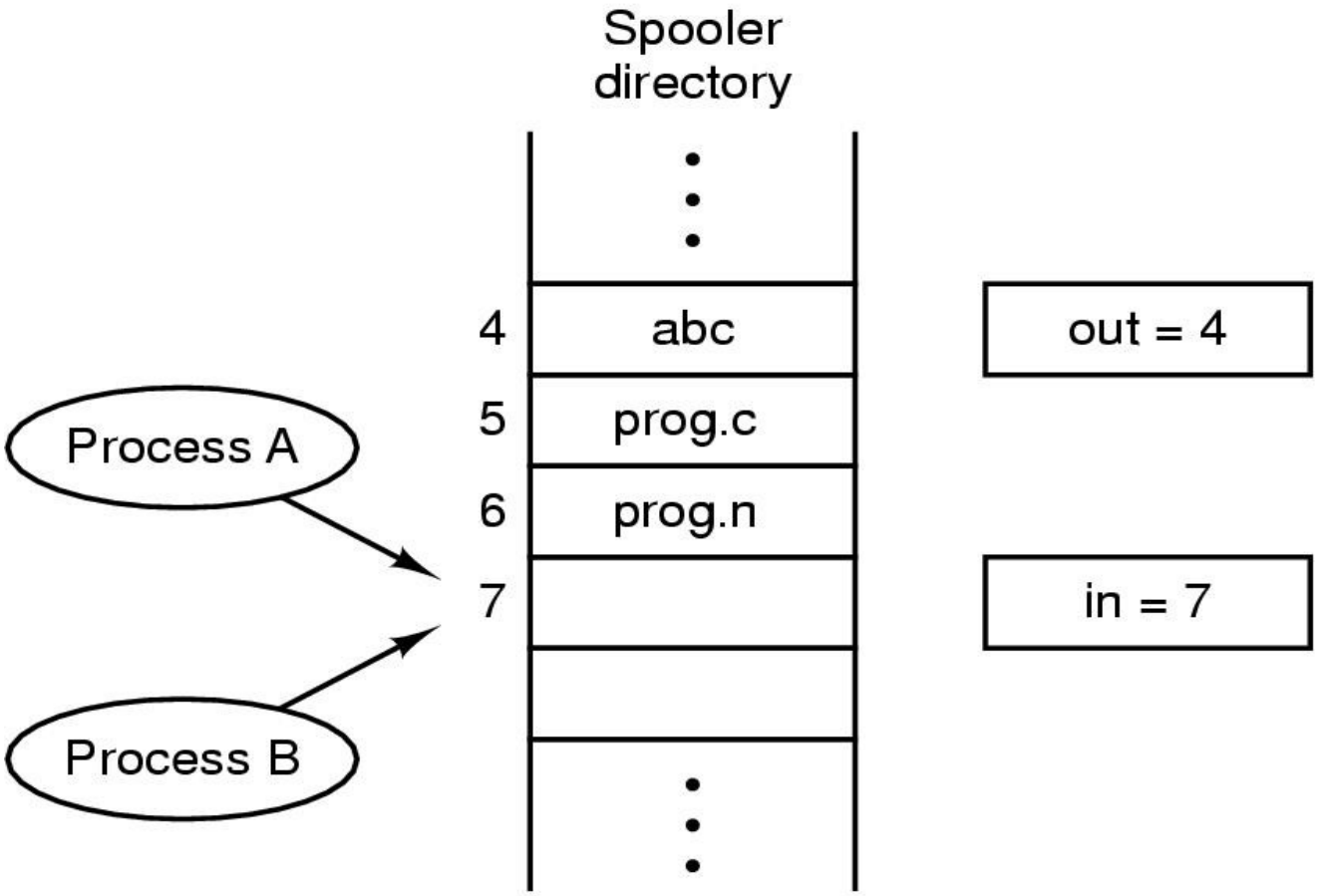
- τη συνεργασία
- τον ανταγωνισμό

Η συνεργασία υλοποιείται με την ύπαρξη μηχανισμών ελέγχου για την αποστολή και παραλαβή δεδομένων και σημάτων.

Ο ανταγωνισμός υλοποιείται με την ύπαρξη μηχανισμών που επιτρέπουν τον διαμοιρασμό πόρων και δεδομένων.

Πολλές φορές η συνεργασία και ο ανταγωνισμός συνυπάρχουν στην επίλυση του ίδιου προβλήματος. Πχ η επικοινωνία μέσω διαμοιραζόμενης μνήμης μπορεί να προκαλέσει συνθήκες ανταγωνισμού ή η αποφυγή ανταγωνισμού απαιτεί συνεργασία.

Συνθήκες ανταγωνισμού (1)



Δύο διεργασίες Α και Β προσπαθούν να προσπελάσουν μοιραζόμενη μνήμη (εδώ το spooler directory) 'ταυτόχρονα'.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Συνθήκες ανταγωνισμού (2)

Process A	Process B	Spooler directory
next_free_slot =in (7)		in=7
Timeout		
	next_free_slot=in (7)	in=7
	Copy_file_in_next_free_slot	
	in=next_free_slot+1	in=8
	Do something else	
	Timeout	
Copy_file_in_next_free_slot		
in=next_free_slot+1		in=8
Do Something else		
Timeout		

Συνθήκες ανταγωνισμού (3)

Οι συνθήκες κατά τις οποίες δύο ή περισσότερες διεργασίες διαμοιράζονται την ίδια περιοχή μνήμης (ή πόρο) και το τελικό αποτέλεσμα εξαρτάται από τη σειρά αλλά και το χρόνο εκτέλεσής τους, λέγονται συνθήκες ανταγωνισμού (race conditions).

Η διόρθωση προγραμμάτων που περιλαμβάνουν συνθήκες ανταγωνισμού δεν είναι εύκολη. Τα σφάλματα παρουσιάζονται τυχαία και μόνο κατά τη διενέργεια αλλεπάλληλων ελέγχων.

Ο έλεγχος του ανταγωνισμού αφορά αναπόφευκτα το λ.σ. διότι αυτό υλοποιεί τον διαμοιρασμό των πόρων και των δεδομένων (διαδιεργασιακή επικοινωνία), άρα οποιαδήποτε λύση θα αφορά και υποστήριξη από το λ.σ.

Επίσης, σε επίπεδο προγραμματισμού εφαρμογών , πρέπει να προσφέρονται λύσεις κλήσεων συστήματος για σύνδρομο (cocurrent) προγραμματισμό, δηλαδή εφαρμογές με πολλά νήματα ή διεργασίες.

Κρίσιμες περιοχές (1)

Για να αποφύγουμε τις συνθήκες ανταγωνισμού πρέπει να επιτύχουμε αμοιβαίο αποκλεισμό (mutual exclusion): όταν μια διεργασία επεξεργάζεται μια διαμοιραζόμενη μεταβλητή τότε η εκτέλεσή της δεν πρέπει να διακοπεί μέχρι να ολοκληρώσει τις ενέργειες που σχετίζονται με αυτή τη μεταβλητή.

Το τμήμα του κώδικα της διεργασίας που αναφέρεται στην επεξεργασία της διαμοιραζόμενης μεταβλητής λέγεται κρίσιμη περιοχή (critical region) ή κρίσιμο τμήμα (critical section).

Οι λειτουργίες μέσα στις κρίσιμες περιοχές πρέπει να εκτελούνται αδιαίρετα, δηλαδή ή πλήρως ή καθόλου.

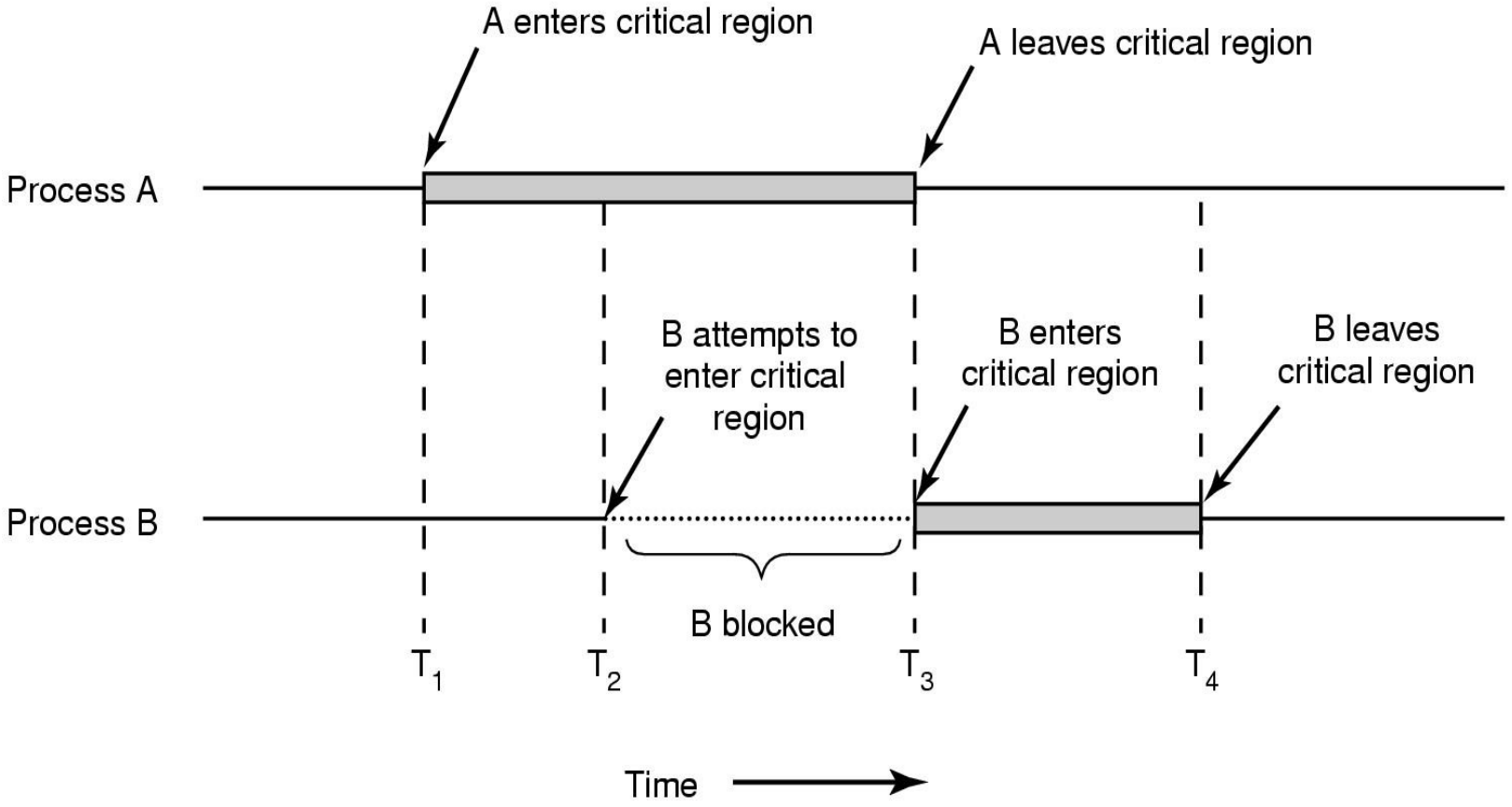
Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Κρίσιμες περιοχές (2)

Συνθήκες αποφυγής συνθηκών ανταγωνισμού:

- Δύο διεργασίες δεν μπορεί να βρίσκονται ταυτόχρονα στις ίδιες κρίσιμες περιοχές τους.
- Δεν επιτρέπονται ιδιαίτερες παραδοχές σχετικά με τη ταχύτητα ή τον αριθμό των επεξεργαστών.
- Διεργασία που δε βρίσκεται σε κρίσιμη περιοχή δεν μπορεί να αναστείλει άλλες διεργασίες.
- Καμία διεργασία δεν θα πρέπει να περιμένει για πάντα για να εισέλθει σε κρίσιμη περιοχή.

Κρίσιμες περιοχές (3)



Αμοιβαίος αποκλεισμός με χρήση κρίσιμων περιοχών.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Αναστολή διακοπών

Η διεργασία που μπαίνει σε κρίσιμη περιοχή αναστέλει τις διακοπές.

Απλή λύση αλλά απαιτεί εκτέλεση privileged instructions από τις διεργασίες (δικαιώματα kernel mode).

Η λύση αυτή χρησιμοποιείται μόνο από τον πυρήνα, όταν πχ εκτελεί context switch και ενημερώνει πίνακες και λίστες διεργασιών, συσκευών, πόρων.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Ενεργός Αναμονή - Λήθαργος και Αφύπνιση

Στην Ενεργό Αναμονή οι διεργασίες που προσπαθούν να εισέλθουν στη κρίσιμη περιοχή εκτελούν ένα συνεχή βρόχο ερώτησης-αναμονής. Απλή υλοποίηση αλλά με μειονέκτημα τη σπατάλη κύκλων CPU για συνεχείς ερωτήσεις.

Πχ Κλειδώματα (Lock, Spinlock)

Στο Λήθαργο και Αφύπνιση οι διεργασίες που προσπαθούν να εισέλθουν στη κρίσιμη περιοχή ρωτούν μια φορά και αν αποτύχουν μεταπίπτουν σε αναστολή αναμένοντας σήμα αφύπνισης. Απαιτούνται πιο σύνθετες υλοποιήσεις.

Πχ Mutexes, Σηματοφόροι (Semaphores)

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Κλείδωμα (Lock, Spinlock) (1)

```
Process 0
...
enter_region(LOCK);
critical_region();
leave_region(LOCK);
...
```

```
Process 1
...
enter_region(LOCK);
critical_region();
leave_region(LOCK);
...
```

Η μεταβλητή τύπου κλειδώματος LOCK ελέγχει την πρόσβαση των διεργασιών 0 και 1 στη κρίσιμη περιοχή. Οι ρουτίνες enter_region() και leave_region() εξηγούνται παρακάτω.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Εντολή TSL (Test and Set Lock)

```
enter_region:
        TSL REGISTER,LOCK       | copy lock to register and set lock to 1
        CMP REGISTER,#0         | was lock zero?
        JNE enter_region        | if it was nonzero, lock was set, so loop
        RET                     | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0            | store a 0 in lock
        RET                     | return to caller
```

Η είσοδος στη κρίσιμη περιοχή εκτελείται με την εντολή TSL η οποία ελέγχει και προσπαθεί να αλλάξει την τιμή του κλειδώματος σε μια αδιαίρετη ενέργεια (atomic action). Η έξοδος από τη κρίσιμη περιοχή επαναφέρει την τιμή αρχική του κλειδώματος.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Lock, Spinlock: παράδειγμα (1)

```
PROCESS 0

a = data; /* a = 10 */
a++;      /* a = 11 */


data = a; /* data = 11 */
```

```
PROCESS 1



b = data; /* b = 10 */
b--;      /* b = 9 */

data = b; /* data = 9 */
```

Η τροποποίηση της μεταβλητής data είναι η κρίσιμη περιοχή.
Πιθανή εκτέλεση χωρίς κλείδωμα προκαλεί σφάλμα.

Lock, Spinlock: παράδειγμα (2)

```
PROCESS 0

lock (&a_lock);


a = data;
a++;
data = a;
unlock (&a_lock);
```

```
PROCESS 1


lock (&a_lock);

/* spinning */
/* spinning */
/* spinning */
/* spinning */
b = data;
b--;
data = b;
unlock (&a_lock);
```

Η τροποποίηση της μεταβλητής data είναι η κρίσιμη περιοχή. Προστατεύεται από το κλείδωμα a_lock.

Mutexes (1)

Δυαδική σηματοφόρος (Binary Semapthore)

Υποστήριξη υλικού (εντολή γλώσσας μηχανής) = mutex.

Έστω s μεταβλητή τύπου mutex.

Αρχικοποίηση s = 1;

P(s) ή down(&s) sleep() until (s ==1) ; s=0; // wakeup(myself)

V(s) ή up(&s) s=1; // wakeup(someone)

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Mutexes (2)

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Υλοποίηση mutex lock και mutex unlock.

Βασική διαφορά με την εντολή TSL το CALL thread_yield, δηλαδή κλήση συστήματος για αναστολή λειτουργίας (sleep()).

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Mutexes και POSIX threads

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Η Pthread_mutex_trylock προσπαθεί μόνο μια φορά και σε περίπτωση αποτυχίας επιστρέφει κωδικό σφάλματος.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Mutexes και POSIX threads: παράδειγμα

```
THREAD 1

pthread_mutex_lock (&mut);

a = data;
a++;
data = a;
pthread_mutex_unlock (&mut);
```

```
THREAD 2


pthread_mutex_lock (&mut);
/* blocked */
/* blocked */
/* blocked */
/* blocked */
b = data;
b--;
data = b;
pthread_mutex_unlock (&mut);
```

Η διαφοροποίηση από τα κλειδώματα βρίσκεται στη κατάσταση Blocked αντί Ready/Running (spinning).

Σηματοφόροι (Semaphores)

Dijkstra (1965):

Ειδικός τύπος δεδομένων, συνδυασμός κλειδώματος-ξεκλειδώματος και ληθάργου-αφύπνισης σε δύο αδιαίρετες ενέργειες (atomic actions).

Έστω s μεταβλητή τύπου semaphore.

Αρχικοποίηση	s = N;	
P(s) ή down(&s)	sleep() until (s > 0); s=s-1;	// wakeup(myself)
V(s) ή up(&s)	s=s+1;	// wakeup(someone)

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

POSIX semaphores

sem_init() -- Αρχικοποιεί ένα σηματοφόρο.

sem_destroy() -- Ακυρώνει ένα σηματοφότο.

sem_getvalue() -- Αντιγράφει τη τιμή του σηματοφόρου σε ακέραιο.

sem_wait(), sem_trywait() -- Αναστέλει τη διεργασία (ή επιστρέφει σήμα) όσο ο σηματοφόρος είναι δεσμευμένος από άλλη διεργασία. Αλλιώς προχωρεί και μειώνει τη τιμή του σηματοφόρου.

sem_post() -- Αύξηση της τιμής του σηματοφόρου.

Παράδειγμα (1)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NITER 1000000
#define SHARED 1
int count = 0;

void * ThreadAdd(void * a)
{
    int i, tmp;
    for(i = 0; i < NITER; i++)
    {
        tmp = count;
        tmp = tmp+1;
        count = tmp;
    }
}

int main(int argc, char * argv[])
{
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, ThreadAdd, NULL);
    pthread_create(&tid2, NULL, ThreadAdd, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    if (count != 2 * NITER)
        printf("\n BOOM! count is [%d], should be %d\n", count, 2*NITER);
    else
        printf("\n OK! count is [%d]\n", count);
    pthread_exit(NULL);
}
```

Παράδειγμα (2)

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#define NITER 1000000
#define SHARED 1
int count = 0;
sem_t mutx;

void * ThreadAdd(void * a)
{
    int i, tmp;
    for(i = 0; i < NITER; i++)
    {
        sem_wait(&mutx);
        tmp = count;
        tmp = tmp+1;
        count = tmp;
        sem_post(&mutx);
    }
}
int main(int argc, char * argv[])
{
    pthread_t tid1, tid2;
    sem_init(&mutx, SHARED, 1);
    pthread_create(&tid1, NULL, ThreadAdd, NULL);
    pthread_create(&tid2, NULL, ThreadAdd, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    ...
    pthread_exit(NULL);
}
```

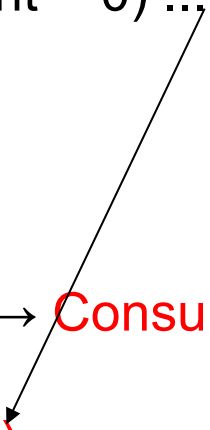
Παραγωγός – Καταναλωτής
Producer - Consumer / Bounded Buffer Problem

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639

Παραγωγός - Καταναλωτής: συνθήκη ανταγωνισμού

Producer	Consumer	Count
	if (count==0) ...Timeout	count=0
...		
insert_item(item)		
count=count+1		count=1
if (count==1) wakeup(consumer) →	Consumer is not sleeping	
... Timeout		
	sleep()	
... After some time		...
...		count=N
if (count==N) sleep()		



Πρέπει να 'προστατευθούν' : οι πράξεις count+/-1 και
οι συνθήκες/γεγονότα if (count==..)sleep()/wakeup()

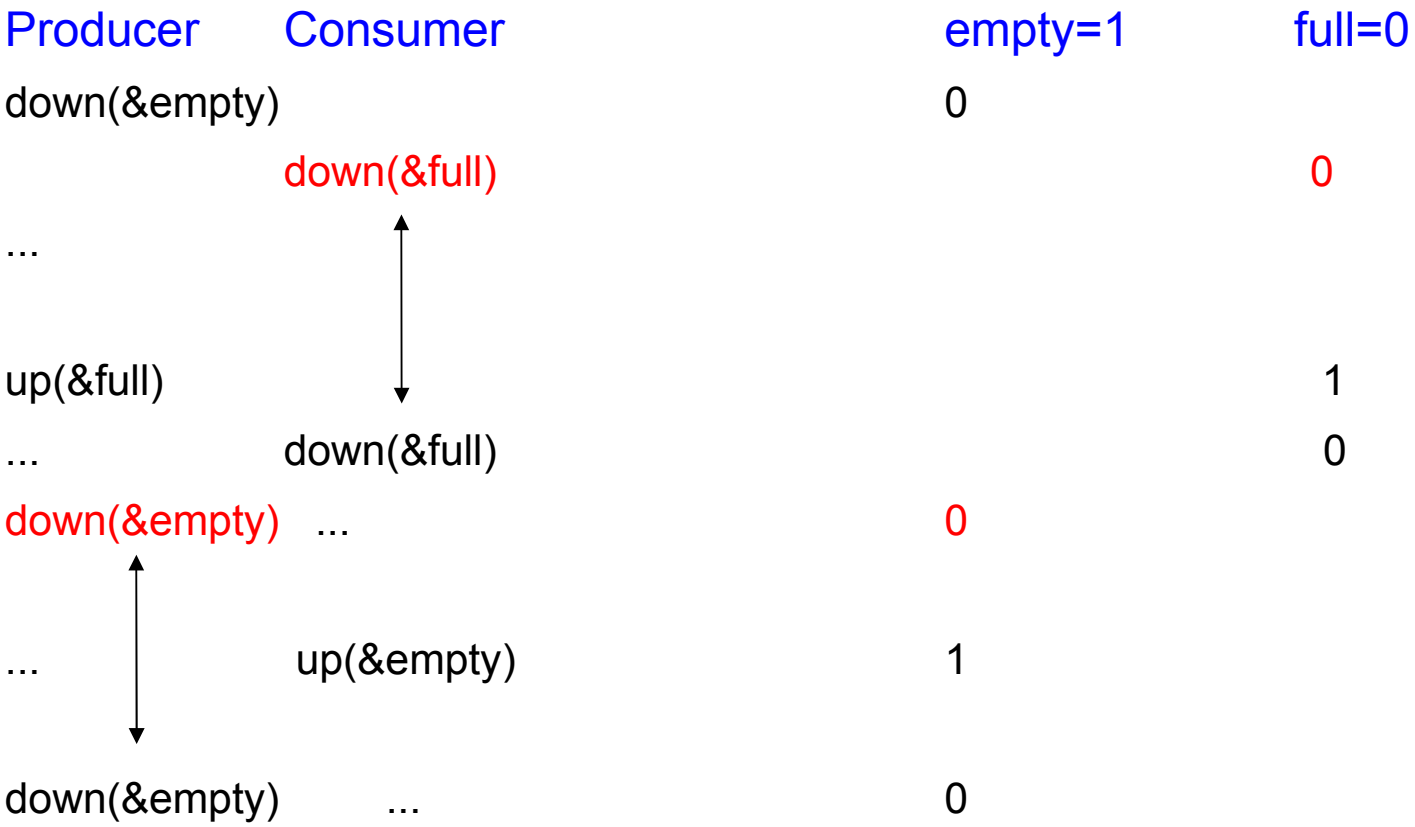
Παραγωγός-Καταναλωτής (1)

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#define SHARED 1

void *Producer(void *);     /* the two threads */
void *Consumer(void *);

sem_t empty, full;    /* the global semaphores */
int data;             /* shared buffer         */
int numIters;

int main(int argc, char *argv[]) { /* thread ids and attributes */
  pthread_t pid, cid;
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  numIters = atoi(argv[1]);
  sem_init(&empty, SHARED, 1);  /* sem empty = 1 */
  sem_init(&full, SHARED, 0);   /* sem full = 0  */

  printf("main started\n");
  pthread_create(&pid, &attr, Producer, NULL);
  pthread_create(&cid, &attr, Consumer, NULL);
  pthread_join(pid, NULL);
  pthread_join(cid, NULL);
  printf("main done\n");
}
```

Παραγωγός- Καταναλωτής (2)

```
/* deposit 1, ..., numIters into the data buffer */
void *Producer(void *arg) {
  int produced;
  printf("Producer created\n");
  for (produced = 0; produced < numIters; produced++) {
    sem_wait(&empty);
    data = produced;
    sem_post(&full);
  }
}

/* fetch numIters items from the buffer and sum them */
void *Consumer(void *arg) {
  int total = 0, consumed;
  printf("Consumer created\n");
  for (consumed = 0; consumed < numIters; consumed++) {
    sem_wait(&full);
    total = total+data;
    sem_post(&empty);
  }
  printf("for %d iterations, the total is %d\n", numIters, total);
}
```

Παράδειγμα εκτέλεσης



Προσπαθήστε να εκτελέσετε διάφορα σενάρια, πχ τι θα συνέβαινε αν η σειρά των down() ή των up() ήταν διαφορετική σε μια από τις δύο διεργασίες;

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Παραγωγός – Καταναλωτής II

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                              /* TRUE is the constant 1 */
        item = produce_item( );                 /* generate something to put in buffer */
        down(&empty);                           /* decrement empty count */
        down(&mutex);                           /* enter critical region */
        insert_item(item);                      /* put new item in buffer */
        up(&mutex);                             /* leave critical region */
        up(&full);                              /* increment count of full slots */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                              /* infinite loop */
        down(&full);                            /* decrement full count */
        down(&mutex);                           /* enter critical region */
        item = remove_item( );                  /* take item from buffer */
        up(&mutex);                             /* leave critical region */
        up(&empty);                             /* increment count of empty slots */
        consume_item(item);                     /* do something with the item */
    }
}
```

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639

Παράδειγμα εκτέλεσης II

Producer	Consumer	mutex=1	empty=N	full=0
down(&empty)			N-1	
down(&mutex)		0		
	down(&full)			0
producing				
up(&mutex)		1		
up(&full)				1
	down(&full)			0
	down(&mutex)	0		
down(&empty)			N-2	
down(&mutex)		0		
	consuming			
	up(&mutex)	1		
	up(&empty)		N-1	
down(&mutex)		0		
producing				
up(&mutex)		1		
up(&full)				1
down(&empty)			N-2	
down(&mutex)		0		
producing				
up(&mutex)		1		
up(&full)				1

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Παρατηρήσεις (1)

Κάθε μεταβλητή (ή δομή δεδομένων) που διαμοιράζεται αυτόνομα προστατεύεται από ένα κλείδωμα ή από σηματοφόρο.

Οι λύσεις είναι ισοδύναμες, με βασική διαφορά την ενεργό αναμονή ή την αναστολή, αντίστοιχα.

Μπορεί να έχουμε κλειδώματα σε ολόκληρες δομές δεδομένων ή μόνο σε στοιχεία τους, πχ κλείδωμα πίνακα, γραμμής ή στοιχείου,

κλείδωμα αρχείου, εγγραφής ή πεδίου. Επομένως μπορεί να έχουμε δομές κλειδωμάτων ή σηματοφόρων.

Ο προγραμματισμός με κλειδώματα και σηματοφόρους χρησιμοποείται κυρίως σε χαμηλό επίπεδο, για καλά ορισμένες δομές και συγκεκριμένες χρήσεις. Η βασική δυσκολία βρίσκεται στον έλεγχο όλων των δυνατών συνδυασμών και χρονικών ακολουθιών αμοιβαίου αποκλεισμού και sleep() - wakepu().

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Παρατηρήσεις (2)

Όταν προγραμματίζουμε σε ψηλότερο επίπεδο του λ.σ., ή για την ανάπτυξη πολυ-διεργασιακών ή πολυ-νηματικών εφαρμογών σε επίπεδο χρήστη, και γενικότερα στον συντρέχοντα (concurrent) προγραμματισμό, χρησιμοποιούμε κλήσεις συστήματος σε ανώτερες δομές χειρισμού της διαδιεργασιακής επικοινωνίας.

- Ελεγκτές (Monitors) και Μεταβλητές Συνθήκης (Condition Variables)
- Μεταβίβαση Μηνυμάτων (Message Passing)
- Σωληνώσεις ή Διαύλους (Pipes)
- Απεικόνιση Μνήμης (Memory Mapping)

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Ελεγκτές και μεταβλητές συνθήκης σε Java Threads (1)

```
synchronized class MonitorProducerConsumer {
    private boolean full, empty;
    private integer count;

    public void insert(Item data) {
        if(count == N) wait(full);
        insert_item(item);
        count++;
        if(count == 1) signal(empty);
    }

    public Item remove() {
        Item data;
        if(count == 0) wait(empty);
        data = remove_item();
        count--;
        if(count == N-1) signal(full);
    }

    ...
}
```

Ελεγκτές και μεταβλητές συνθήκης σε Java Threads (2)

```
static MonitorProducerConsumer theMonitor
    = new MonitorProducerConsumer();

public class Producer extends Thread {
    public void run() {
        while(true) {
            Item data = produce_item();
            theMonitor.insert(data);
        }
    }
    ...
}

public class Consumer extends Thread {
    public void run() {
        while(true) {
            Item data = theMonitor.remove(data);
            consume_item(data);
        }
    }
    ...
}
```

Μεταβλητές συνθήκης με mutexes (1)

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mtx;
pthread_cond_t cond;

int how_many = 10;
int pool = 0;

void * producer(void * ptr)
{
    while (how_many > 0)
    {
        pthread_mutex_lock(&mtx);
        printf("producer: %d\n", how_many);
        pool = how_many;
        how_many--;
        pthread_mutex_unlock(&mtx);
        pthread_cond_signal(&cond);
    }
    pthread_exit(0);
}
```

Μεταβλητές συνθήκης με mutexes (2)

```
void * consumer(void * ptr)
{
    while (how_many > 0)
    {
        pthread_mutex_lock(&mtx);
        pthread_cond_wait(&cond, &mtx);
        printf("consumer: %d\n", pool);
        pool = 0;
        pthread_mutex_unlock(&mtx);
    }
    pthread_exit(0);
}

int main(int argc, char ** argv)
{
    pthread_t prod, cons;
    pthread_mutex_init(&mtx, 0);
    pthread_cond_init(&cond, 0);
    pthread_create(&cons, 0, consumer, 0);
    pthread_create(&prod, 0, producer, 0);
    pthread_join(prod, 0);
    pthread_join(cons, 0);
    pthread_cond_destroy(&cond);
    pthread_mutex_destroy(&mtx);
    return 0;
}
```

Ελεγκτές και μεταβλητές συνθήκης

Ο ελεγκτής μπορεί να υλοποιηθεί με τους μηχανισμούς που ήδη συζητήθηκαν, αλλά είναι διαφανής στο χρήστη.

Χρήση μεταβλητών συνθήκης (condition variables)

- wait (s) : sleep() until someone.signal(s)
- signal(s) : wakeup(someone.waiting(s))

Κανόνας: η signal() πάντα τελευταία εντολή στη διαδικασία.

Διαφορά με σηματοφόρους: το s είναι boolean (binary) δεν λαμβάνει τιμές, άρα δεν έχει 'μνήμη'.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Μεταβίβαση Μηνυμάτων (1)

Κλήσεις συστήματος για ανάγνωση-εγγραφή σε ουρές μηνημάτων (message queues) τις οποίες διαχειρίζεται το λ.σ.

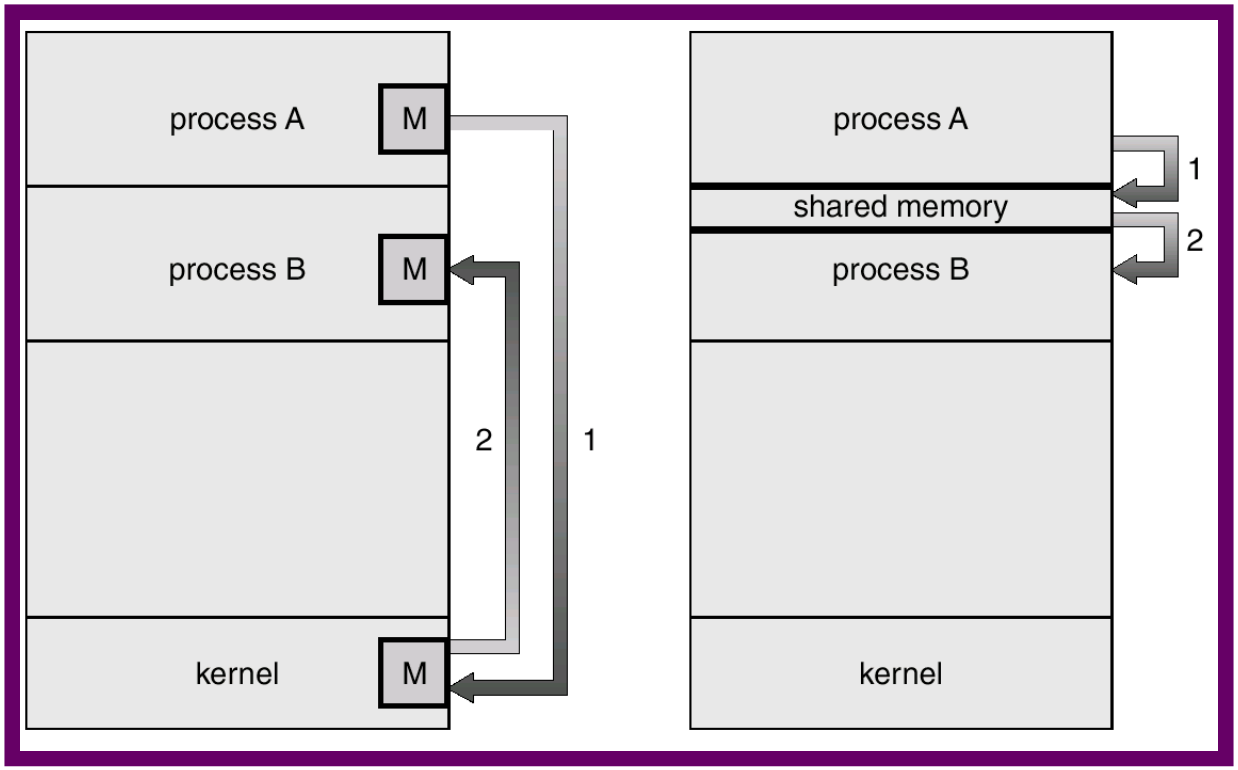
Send(to_pid, &data);
Receive(from_pid, &data);

Δεν υπάρχουν προβλήματα διαμοιρασμού μνήμης, η μέθοδος μπορεί εύκολα να επεκταθεί σε απομακρυσμένα συστήματα (πχ client-server)

Απαιτείται:

Εγκαθίδρυση σύνδεσης και πιστοποίηση ταυτότητας.
Ενημέρωση αποστολής δεδομένων με κάποιο σήμα στον παραλήπτη.
Σύνθεση, αποσύνθεση μυνημάτων (κάποια δομή).
Επιβεβαίωση παραλαβής και αρίθμηση ακολουθίας μηνυμάτων.

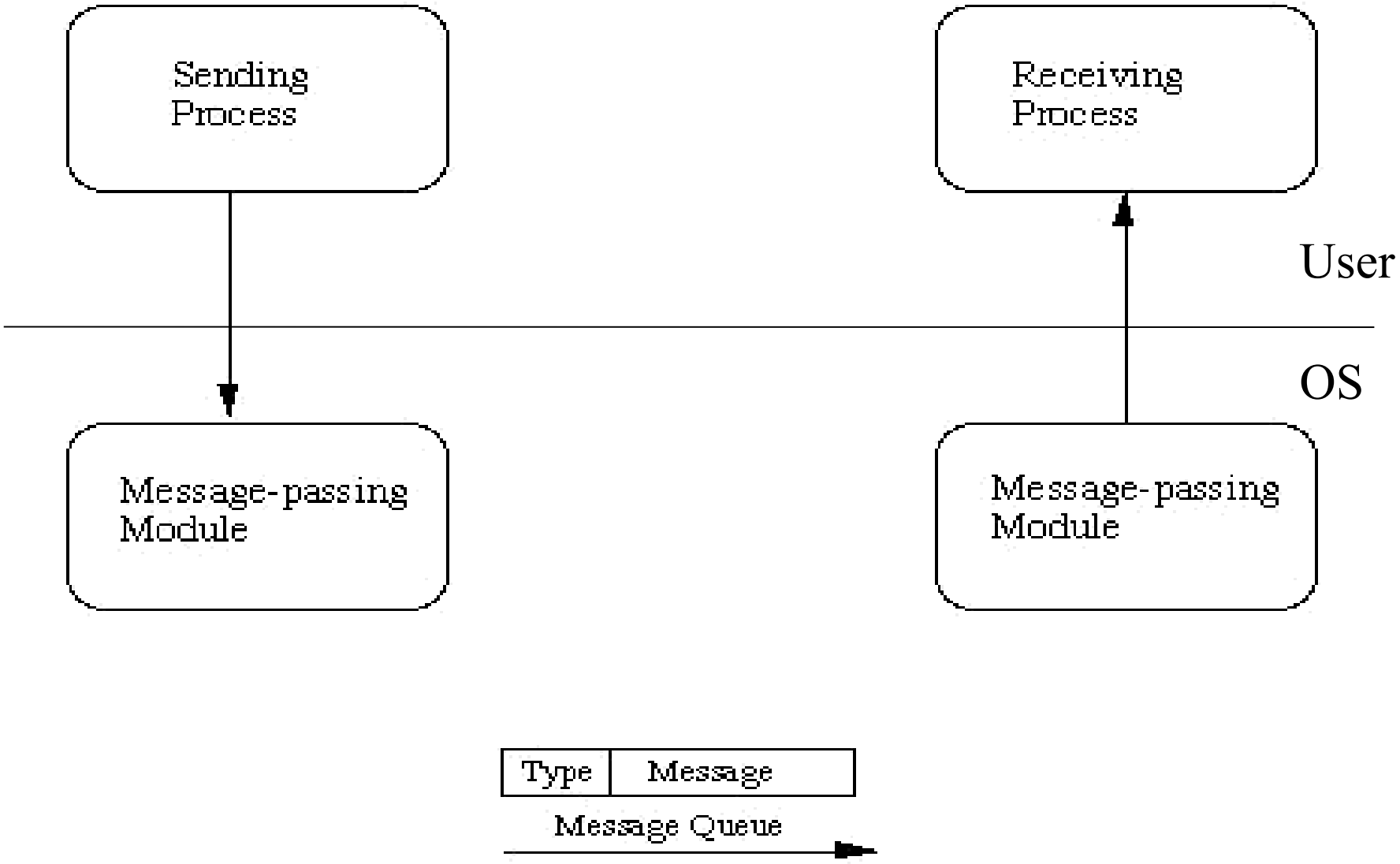
Μεταβίβαση Μηνυμάτων (2)



Μεταβίβαση Μηνυμάτων Μοιραζόμενη Μνήμη

Διαφοροποίηση κυρίως από τη πλευρά των κλήσεων συστήματος (API).
Από τη πλευρά του λ.σ. καταλήγουμε σε παρόμοια αντιμετώπιση:
Κοινή προυπόθεση όλων των λύσεων: μηχανισμός κοινής πρόσβασης.

Μεταβίβαση Μηνυμάτων (3)



Υλοποίηση μεταβίβασης μηνυμάτων

Ζητήματα σχεδίασης του λ.σ.

- Άμεση ή έμμεση επικοινωνία (μέσω mailbox).
- Απλή ή πολλαπλή επικοινωνία (multicasting).
- Μια ή πολλές συνδέσεις ανάμεσα σε δύο διεργασίες.
- Μέγεθος ενταμιευτή ή mailbox (περιορισμένο ή όχι).
- Μέγεθος και δομή μηνύματος.
- Σταθερό ή μεταβλητό μέγεθος μηνύματος;
- Μονόδρομη ή αμφίδρομη σύνδεση.
- Ανασταλτική (Blocking) ή σύγχρονη (synchronous) --
- Μη-ανασταλτική (non-blocking) ή ασύγχρονη (asynchronous)

POSIX message queues

mq_open( )	Opens a message queue.
mq_close( )	Closes a message queue.
mq_unlink( )	Removes a message queue.
mq_send( )	Sends a message to a queue.
mq_receive( )	Gets a message from a queue.
mq_notify( )	Signals a task that a message is waiting on a queue.
mq_setattr( )	Sets a queue attribute.
mq_getattr( )	Gets a queue attribute.

Παραγωγός Καταναλωτής με Μεταβίβαση Μηνυμάτων (1)

```
#include <pthread.h>
#include <mqueue.h>

int main(void)
{
    mqd_t messageQueueDescr;

    /* Create the message queue for sending information between tasks. */
    messageQueueDescr = mq_open(messageQueuePath,
                                (O_CREAT | O_EXCL | O_RDWR));

    /* Create the producer task using the default task attributes. Do not
     * pass in any parameters to the task. */
    pthread_create(&producerTaskObj, NULL, (void *)producerTask, NULL);

    /* Create the consumer task using the default task attributes. Do not
     * pass in any parameters to the task. */
    pthread_create(&consumerTaskObj, NULL, (void *)consumerTask, NULL);

    /* Allow the tasks to run. */
    pthread_join(producerTaskObj, NULL);
    pthread_join(consumerTaskObj, NULL);

    return 0;
}
```

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

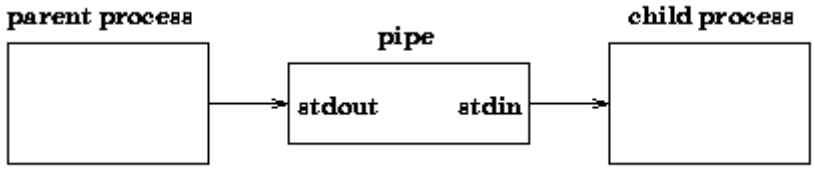
Παραγωγός Καταναλωτής με Μεταβίβαση Μηνυμάτων (2)

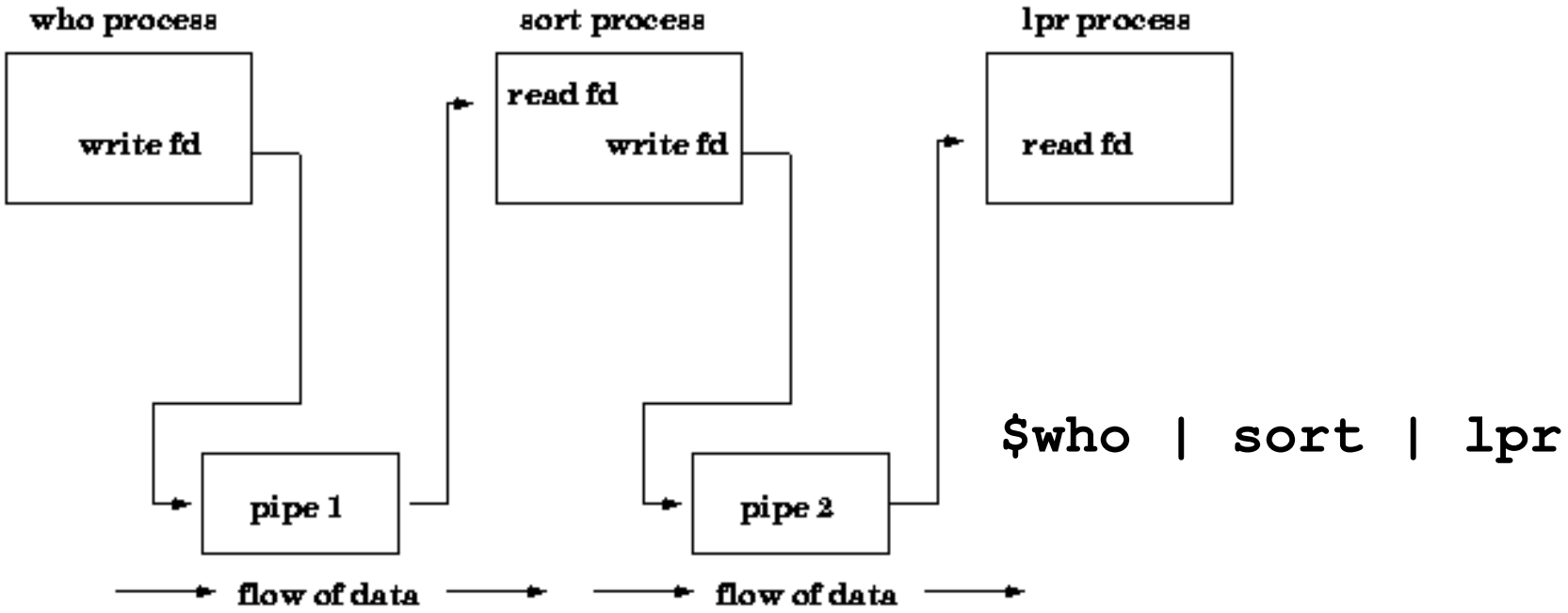
```
void producerTask(void *param)
{
    mqd_t messageQueueDescr;
    msgbuf_t msg;
    messageQueueDescr = mq_open(messageQueuePath, O_WRONLY);
    while (1)
    {
            /* Create message msg */
            /* Send message */
            mq_send(messageQueueDescr, &msg.buf, sizeof(msgbuf_t), 0);
        }
    }
}
void consumerTask(void *param)
{
    mqd_t messageQueueDescr;
    msgbuf_t rcvMsg;
    messageQueueDescr = mq_open(messageQueuePath, O_RDONLY);
    while (1)
    {
        /* Wait for a new message. */
        mq_receive(messageQueueDescr, &rcvMsg.buf, sizeof(msgbuf_t),
                                                              NULL);
        /* Cosnume message msg */
    }
}
```

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Δίαυλος UNIX (pipe)

```
$parent | child
```






```
$who | sort | lpr
```

Η ουρά μηνυμάτων είναι ένα αρχείο (κειμένου ή δυαδικό).

Δίαυλος UNIX (pipe)

```
#include <stdio.h>
#include <string.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{ int p[2];
  int i, pid, status;
  char buffer[20];
    pipe(p);                          /* setting up the pipe   */
    if ((pid = fork()) == 0)
    /* in child */
    {  close(p[1]);                   /* child closes p[1]     */
       while ((i = read(p[0], buffer, 8)) != 0)
       { buffer[i] = '\0';            /* string terminator     */
         printf("%d chars :%s: received by child\n", i, buffer);
       }
       close(p[0]);
       exit(0);                      /* child terminates      */
    }
    /* in parent */
    close(p[0]);
    /* parent writes p[1]    */
    write(p[1], "Hello there, from me?", sizeof("Hello there, from me?"));
    write(p[1], "How are you? - Ercal", sizeof("How are you? - Ercal"));
    close(p[1]);                      /* finished writing p[1] */
    while (wait(&status)!=pid);       /* waiting for pid       */
    if (status == 0)  printf("child finished\n");
    else printf("child failed\n");
    return(0);
}
```

Απεικόνιση Μνήμης (Memory Mapping, mmap)

Κλήσεις συστήματος σε UNIX/POSIX που επιτρέπουν τη χρήση διαμοιραζόμενης μνήμης για το διαμοιρασμό αρχείων μεταξύ διεργασιών.

Ο ενταμιευτής (buffer) ενός αρχείου, διαμοιράζεται μεταξύ δύο ή περισσότερων διεργασιών. Για τον έλεγχο του διαμοιρασμού χρησιμοποιούνται σηματοφόροι ή κλειδώματα.

Το API αυτό επιτρέπει και το διαμοιρασμό δεδομένων ή κοινών βιβλιοθηκών που βρίσκονται στη δευτερεύουσα μνήμη λόγω ιδεατής μνήμης (πχ σελίδες από χώρο διευθύνσεων διεργασίας).

Κλήσεις Win32 για διαχείριση διεργασιών

Win32 API Function	Description
CreateProcess	Create a new process
CreateThread	Create a new thread in an existing process
CreateFiber	Create a new fiber
ExitProcess	Terminate current process and all its threads
ExitThread	Terminate this thread
ExitFiber	Terminate this fiber
SetPriorityClass	Set the priority class for a process
SetThreadPriority	Set the priority for one thread
CreateSemaphore	Create a new semapahore
CreateMutex	Create a new mutex
OpenSemaphore	Open an existing semaphore
OpenMutex	Open an existing mutex
WaitForSingleObject	Block on a single semaphore, mutex, etc.
WaitForMultipleObjects	Block on a set of objects whose handles are given
PulseEvent	Set an event to signaled then to nonsignaled
ReleaseMutex	Release a mutex to allow another thread to acquire it
ReleaseSemaphore	Increase the semaphore count by 1
EnterCriticalSection	Acquire the lock on a critical section
LeaveCriticalSection	Release the lock on a critical section