

Εργαστήριο Λειτουργικών Συστημάτων

Μάθημα 6^{ου} Εξαμήνου,

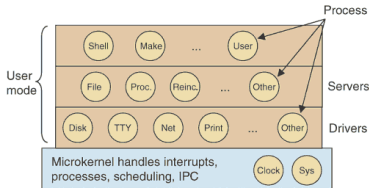
Τομέας Λογικού και Υπολογιστών

Ιωάννης Χατζηγιαννάκης

Τετάρτη, 24 Μαρτίου, 2010
Αίθουσα ΒΑ



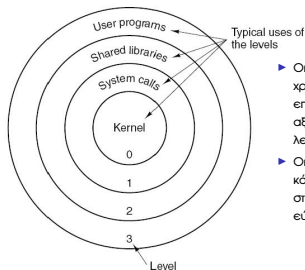
Η Εσωτερική Δομή του Minix 3



- ▶ Οι λειτουργίες που προσφέρει το Λ.Σ. υλοποιούνται ως ανεξάρτητες διεργασίες
- ▶ Η επικοινωνία μεταξύ των διεργασιών γίνεται με την ανταλλαγή μηνυμάτων



Ιεράρχηση των επιπέδων του Minix 3



- ▶ Οι 'απλές' διεργασίες' χρησιμοποιούν το 2ο επίπεδο για να αξιοποιήσουν τις λειτουργίες του Λ.Σ.
- ▶ Οι βιβλιοθήκες του Λ.Σ. κάνουν την πρόσβαση στο 1ο επίπεδο 'ποιο εύχρηστη'



Επικοινωνία με το System Task

- ▶ Οι διαχειριστές επικοινωνούν μεταξύ τους, με τους οδηγούς και με τον πυρήνα **με την χρήση μηνυμάτων**
- ▶ Το System Task λαμβάνει όλα τα μηνύματα που αφορούν τον πυρήνα
 - ▶ Είναι το σημείο εισόδου / εξόδου στον πυρήνα για τις διεργασίες υψηλότερων επιπέδων
- ▶ Θα μπορούσαμε να ονομάσουμε όλες αυτές τις αιτήσεις ως κλήσεις του συστήματος
 - ▶ Στο Minix τις ονομάζουμε **κλήσεις του πυρήνα**
- ▶ Δεν είναι προσβάσιμες στις 'απλές' διεργασίες
- ▶ Στις περισσότερες περιπτώσεις μια κλήση του συστήματος μετατρέπεται σε κλήση του πυρήνα με παρόμοιο όνομα
- ▶ Απλά διότι σχεδόν όλες οι κλήσεις πρέπει να τις χειριστεί (σε κάποιο βαθμό, έστω μικρό) ο πυρήνας



Οργάνωση του System Task

- ▶ Η βασική δομή του System Task είναι απλή (αρχείο *kernel/system.c*)
- ▶ Αρχικοποιεί τις εσωτερικές δομές (συνάρτηση *initialize*)
- ▶ Εκτελεί ένα ατέρμονο βρόχο (συνάρτηση *sys_task*)
 - ▶ Δέχεται νέα μηνύματα
 - ▶ Εκτελεί τις αντίστοιχες συναρτήσεις
 - ▶ Απαντάει με τα αποτελέσματα
- ▶ Περιέχει ορισμένες βοηθητικές συναρτήσεις
- ▶ Το System Task αναγνωρίζει 28 τύπου μηνυμάτων
 - ▶ κάθε ένα από αυτά μπορεί να θεωρηθεί μια κλήση του πυρήνα
- ▶ Σε ορισμένες περιπτώσεις ο ίδιος τύπος μηνύματος εμφανίζεται με διαφορετικά ονόματα με την χρήση *macros*
- ▶ Σε άλλες περιπτώσεις μια συγκεκριμένη συνάρτηση επεξεργάζεται διαφορετικούς τύπους μηνυμάτων



Δομή κώδικα του System Task

- ▶ Το System Task υλοποιείται από το *system.h* και *system.c*
 - ▶ Τα αρχεία είναι στον φάκελο */usr/src/kernel*
- ▶ Το Minix δεν είναι **μόνο** για γενική χρήση
 - ▶ Επιπρέπει την ενεργοποίηση/απενεργοποίηση των κλήσεων του πυρήνα – μείωση μεγέθους εκτελέσιμου
 - ▶ Το αρχείο *kernel/config.h* ορίζει ποιες κλήσεις είναι ενεργές
 - ▶ Αν αφαιρέσουμε μια δήλωση, όλα τα τμήματα του κώδικα που σχετίζονται με την συγκεκριμένη κλήση δεν θα γίνουν *compile*
- ▶ Όλες οι κλήσεις του πυρήνα ορίζονται στο αρχείο *com.h*
 - ▶ Βρίσκεται στον φάκελο */usr/src/include/minix*
 - ▶ Αν θέλουμε να προσθέσουμε μια νέα κλήση, πρέπει να την ορίσουμε εδώ
 - ▶ Ανεξάρτητα από το αν θα την απενεργοποιήσουμε ή όχι



Αρχείο Επικεφαλίδας System Task

- ▶ Το αρχείο *system.h* ορίζει τις συναρτήσεις που χειρίζονται κάθε κλήση του πυρήνα
 - ▶ Για την κλήση *sys_XXX* αντιστοιχεί την συνάρτηση *do_XXX*

```
/* Default handler for unused kernel calls. */
_PROTOTYPE( int do_unused, (message *m_ptr) );
_PROTOTYPE( int do_exec, (message *m_ptr) );
_PROTOTYPE( int do_fork, (message *m_ptr) );
_PROTOTYPE( int do_trace, (message *m_ptr) );
_PROTOTYPE( int do_nice, (message *m_ptr) );
_PROTOTYPE( int do_copy, (message *m_ptr) );
#define do_vircopy do_copy
#define do_physcopy do_copy
```

- ▶ Η συνάρτηση *do_unused* είναι μια 'κενή' συνάρτηση
- ▶ Η συνάρτηση *do_copy* αντιστοιχεί επίσης στις κλήσεις *sys_vircopy* *sys_physcopy*



Αντιστοίχιση Συναρτήσεων – Κλήσεις Πυρήνα

- ▶ Για την αντιστοίχιση συναρτήσεων με κλήσεις συστήματος, στο *system.c* ορίζεται ο πίνακας *call_vec[]*

```
PUBLIC int (*call_vec[NR_SYS_CALLS])(message *m_ptr)
```

- ▶ Η αρχικοποίηση γίνεται με την συνάρτηση *initialize* (αρχείο *system.c*)
- ▶ Προσθέτουμε στοιχεία στον πίνακα με την χρήση του *macro map*

```
PRIVATE void initialize(void){
    for (i=0; i<NR_SYS_CALLS; i++){
        call_vec[i] = do_unused;

        /* Process management. */
        map(SYS_FORK, do_fork);
        map(SYS_NICE, do_nice);
        ...
    }
}
```



```
PUBLIC void sys_task(){
    initialize();

    while (TRUE) {
        receive(ANY, &m);
        call_nr = (unsigned) m.m_type - KERNEL_CALL;
        caller_ptr = proc_addr(m.m_source);

        if (!(priv(caller_ptr)->s_call_mask & (1<<call_nr)))
            result = ECALLDENIED;
        else if (call_nr >= NR_SYS_CALLS)
            result = EBADREQUEST;
        else
            result = (*call_vec[call_nr])(&m);
    }
}
```



Προηγούμενο Μάθημα

To System Task
Οργάνωση του System Task
Υλοποίηση του System Task

Λειτουργικό Σύστημα Minix

Διαχειριστής Διεργασιών – Κλήσεις Συστήματος
Ροή Εκτέλεσης Κλήσης Συστήματος
Θέματα Σχεδιασμού

Σύνοψη Μαθήματος

Σύνοψη Μαθήματος
Δεύτερη Άσκηση
Επόμενη Διάλεξη



- ▶ Ο τρόπος που χειρίζονται οι διαχειριστές (managers) τις κλήσεις του συστήματος (system calls) μαζί με τον τρόπο που χειρίζεται το system task τις κλήσεις του πυρήνα (kernel calls)
 - ▶ Κατά την εκκίνηση αρχικοποιεί τις εσωτερικές δομές
 - ▶ Αναθέτει μια συνάρτηση (handler) για κάθε κλήση του συστήματος
 - ▶ Χρησιμοποιεί ένα ατέρμονο βρόγχο για την εξυπηρέτηση των κλήσεων όπου περιμένει για μηνύματα
- ▶ Σε αντίθεση με το system task οι διαχειριστές είναι αυτόνομα εκτελέσιμα προγράμματα
 - ▶ Υλοποιούν την συνάρτηση *main()*
 - ▶ Το εκτελέσιμο αρχείο ενσωματώνεται στο image του συστήματος
- ▶ Σε αντίθεση με το system task οι συναρτήσεις handlers δεν ορίζονται δυναμικά
 - ▶ Η ανάθεση γίνεται κατά το compilation



- ▶ Ο κώδικας των διαχειριστών βρίσκεται στον φάκελο */usr/src/servers*
 - ▶ Κάθε διαχειριστής χρησιμοποιεί ξεχωριστό υποφάκελο π.χ. Ο κώδικας του διαχειριστή διεργασιών είναι στον φάκελο */usr/src/servers/pm*
- ▶ Οι διαχειριστές υλοποιούνται στο αρχείο *main.c*
- ▶ Οι βιβλιοθήκες που χρησιμοποιεί ένας διαχειριστής ορίζεται στο αρχείο *<manager-name>.h*, π.χ. *pm.h*
- ▶ Ειδικές δομές των διαχειριστών ορίζονται στο αρχείο *type.h*
- ▶ Οι πίνακες που χρησιμοποιούν οι διαχειριστές ορίζονται στο αρχείο *table.c*
 - ▶ Σε αυτό το αρχείο ορίζονται οι handlers για κάθε κλήση του συστήματος (system call)
- ▶ Οι συναρτήσεις handlers ορίζονται στο αρχείο *proto.h*

Ορισμός Κλήσεων Συστήματος

- ▶ Όλες οι κλήσεις του συστήματος ορίζονται στο αρχείο `callnr.h`
 - ▶ Βρίσκεται στον φάκελο `/usr/src/include/minix`

```
/* number of system calls allowed */
#define NCALLS          95

#define EXIT            1
#define FORK           2
#define READ           3
#define WRITE          4
#define OPEN           5
#define CLOSE          6
#define WAIT           7
```

- ▶ Αν θέλουμε να προσθέσουμε μια νέα κλήση, πρέπει να την ορίσουμε εδώ



Αρχείο Επικεφαλίδας `proto.h`

- ▶ Το αρχείο `proto.h` ορίζει τις συναρτήσεις που χειρίζονται κάθε κλήση του συστήματος
 - ▶ Για τον συγκεκριμένο διαχειριστή
 - ▶ Για την κλήση `XXX` αντιστοιχεί την συνάρτηση `do_xxx`

```
/* Default handler for unused system calls. */
_PROTOTYPE( int no_sys, (void) );

/* exec.c */
_PROTOTYPE( int do_exec, (void) );

/* forkexit.c */
_PROTOTYPE( int do_fork, (void) );
_PROTOTYPE( int do_pm_exit, (void) );
_PROTOTYPE( int do_waitpid, (void) );
```

- ▶ Η συνάρτηση `no_sys` είναι μια 'κενή' συνάρτηση



Αντιστοίχιση Συναρτήσεων – Κλήσεις Συστήματος

- ▶ Για την αντιστοίχιση συναρτήσεων με κλήσεις συστήματος, στο `table.c` ορίζεται ο πίνακας `call_vec[]`

```
_PROTOTYPE( int (*call_vec[NCALLS]), (void) ) = {
    no_sys,          /* 0 = unused */
    do_pm_exit,     /* 1 = exit */
    do_fork,         /* 2 = fork */
    no_sys,          /* 3 = read */
    ...
};
```

- ▶ Ανατρέχουμε στον πίνακα με βάση τον αριθμό της κλήσης (όπως ορίζεται στο μήνυμα) – αυτή η τεχνική χρησιμοποιείται και σε άλλα σημεία του Minix
- ▶ Η προσθήκη νέων στοιχείων απαιτεί recompilation του συγκεκριμένου διαχειριστή



Εκκίνηση Διαχειριστή Διεργασιών

- ▶ Η εκτέλεση του διαχειριστή γίνεται όπως οποιοδήποτε 'απλό' εκτελέσιμο αρχείο

```
PUBLIC int main() {
    /* Main routine of the process manager. */
    int result, s, proc_nr;
    struct mproc *rmp;
    sigset_t sigset;

    /* initialize process manager tables */
    pm_init();

    /* Main loop: get work, do it, forever */
    while (TRUE) {
        ...
    }
}
```



Αρχικοποίηση Διαχειριστή Διεργασιών

- ▶ Η αρχικοποίηση γίνεται με την συνάρτηση `pm_init()` (αρχείο `main.c`)

```
FORWARD _PROTOTYPE( void pm_init, (void) );
```

- ▶ Συλλέγει στοιχεία για την διαθέσιμη μνήμη και την τρέχουσα χρήση (`image`)
- ▶ Αρχικοποιεί τον πίνακα `mem_chunks` όπου τοποθετούνται τα στοιχεία για την τρέχουσα χρήση
 - ▶ Στο `minix` ο διαχειριστής διεργασιών, διαχειρίζεται και την μνήμη
- ▶ Αρχικοποιεί την λίστα οπών (`hole list`, άδεια τμήματα μνήμης)

```
struct mem_map mem_map[NR_LOCAL_SEGS];  
struct memory mem_chunks[NR_MEMS];
```

- ▶ Αρχικοποιεί το τμήμα του πίνακα των διεργασιών που τηρεί ο διαχειριστής
 - ▶ Ορίζεται στο αρχείο `mproc.h`



Πίνακας Διεργασιών του Διαχειριστή Διεργασιών

```
EXTERN struct mproc {  
    /* points to text, data, stack */  
    struct mem_map mp_seg[NR_LOCAL_SEGS];  
    char mp_exitstatus; /* status when process exits */  
    /* storage for signal # for killed procs */  
    char mp_sigstatus;  
    pid_t mp_pid; /* process id */  
    int mp_endpoint; /* kernel endpoint id */  
    pid_t mp_procgrp; /* pid of process group */  
    pid_t mp_wpid; /* pid process is waiting for */  
    int mp_parent; /* index of parent process */  
  
    /* Child user and system times */  
    clock_t mp_child_utime; /* cumulative user time */  
    clock_t mp_child_stime; /* cumulative sys time */  
    ...  
}
```



Παραλαβή Μηνυμάτων

- ▶ Η παραλαβή των μηνυμάτων (κλήσεις συστήματος που αφορούν τον διαχειριστή) γίνεται με την χρήση της συνάρτησης `get_work()`

```
PRIVATE void get_work() {  
    /* Wait for the next message */  
    if (receive(ANY, &m_in) != OK)  
        panic(__FILE__, "PM receive error", NO_NUM);  
  
    /* extract useful information from it. */  
    who_e = m_in.m_source; /* who sent the message */  
    if (pm_isokendpt(who_e, &who_p) != OK)  
        panic(__FILE__, "PM got invalid endpoint", who_e);  
    call_nr = m_in.m_type; /* system call number */  
  
    /* Process slot of caller */  
    mp = &mproc[who_p < 0 ? PM_PROC_NR : who_p];  
}
```



Βασική Λειτουργία του Διαχειριστή Διεργασιών

```
while (TRUE) {  
    get_work(); /* wait for an PM system call */  
  
    if (call_nr == SYN_ALARM) {  
        ... /* don't reply */  
    } else if (call_nr == SYS_SIG) {  
        ... /* don't reply */  
    } /* check if the system call number is valid */  
    } else if ((unsigned) call_nr >= NCALLS) {  
        result = ENOSYS;  
    } else {  
        /* perform the call */  
        result = (*call_vec[call_nr])();  
    }  
    ...  
}
```



- Ο ορισμός της δομής των μηνυμάτων βρίσκεται στο αρχείο `/usr/src/include/minix/ipc.h`
- Συνολικά 7 διαφορετικοί τύποι μηνυμάτων

```
typedef struct {int m111, m112, m113; char *m1p1, *m1p2, *m1p3;} mess_1;
typedef struct {int m211, m212, m213; long m211, m212; char *m2p1;} mess_2;
typedef struct {int m311, m312; char *m3p1; char m3cal[M3_STRING1];} mess_3;
typedef struct {long m411, m412, m413, m414, m415;} mess_4;
typedef struct {short m5c1, m5c2; int m511, m512; long m511, m512, m513;} mess_5;
typedef struct {int m711, m712, m713, m714; char *m7p1, *m7p2;} mess_7;
typedef struct {int m811, m812; char *m8p1, *m8p2, *m8p3, *m8p4;} mess_8;

typedef struct {
    int m_source; // who sent the message
    int m_type; // what kind of message is it
    union {
        mess_1 m_m1;
        mess_2 m_m2;
        mess_3 m_m3;
        mess_4 m_m4;
        mess_5 m_m5;
        mess_7 m_m7;
        mess_8 m_m8;
    } m_u;
} message;
```



m_source	m_source	m_source	m_source	m_source	m_source	m_source
m_type	m_type	m_type	m_type	m_type	m_type	m_type
m1_11	m2_11	m3_11	m4_11	m5_c1m5_c2	m7_11	m8_11
m1_12	m2_12	m3_12	m4_12	m5_11	m7_12	m8_12
m1_13	m2_13	m3_p1	m4_13	m5_12	m7_13	m8_p1
m1_p1	m2_11		m4_14	m5_11	m7_14	m8_p2
m1_p2	m2_12	m3_oa1	m4_15	m5_12	m7_p1	m8_p3
m1_p3	m2_p1			m5_13	m7_p2	m8_p4

- Το μέγεθος προκύπτει από την αρχιτεκτονική του συστήματος
- Το σχήμα απεικονίζει τις δομές για αρχιτεκτονική 32-bit



Παράδειγμα Κλήσης Συστήματος

- Ας εξετάσουμε την ροή εκτέλεσης μιας κλήσης συστήματος
 - Ένα παράδειγμα με την χρήση της `fork()`
 - Στο 5ο μάθημα επικειρήσαμε κάτι αντίστοιχο για την εντολή `mkdir`

```
#include <string.h>
main() {
    int pid = 0;
    printf("hello world\n");
    pid = fork();
    if (pid == 0) { // Child process executes here
        printf("this is the child\n");
    } else { // parent process executes here
        printf("this is the parent\n");
    }
    printf("bye\n");
    exit();
}
```



Βοηθητική Συνάρτηση για την Κλήση Συστήματος fork (1)

- Η επικοινωνία μεταξύ των διεργασιών γίνεται μέσω ανταλλαγής μηνυμάτων
 - Είτε πρόκειται για επικοινωνία μεταξύ απλών διεργασιών (εργασίες μαθήματος ΛΣ.Ι)
 - Είτε πρόκειται για επικοινωνία με τους διαχειριστές (το παράδειγμα μας)
 - Είτε πρόκειται για επικοινωνία των διαχειριστών με τον πυρήνα (προηγούμενο μάθημα)
- Για λόγους ευκολίας, το σύστημα προσφέρει συναρτήσεις που 'κρύβουν' τους μηχανισμούς ανταλλαγής μηνυμάτων
- Η βοηθητική συνάρτηση δημιουργεί ένα νέο μήνυμα (που περιέχει τις παραμέτρους της κλήσης του συστήματος) και στέλνει το μήνυμα στον αντίστοιχο διαχειριστή/οδηγό
- Συγκεκριμένα για την `fork()` η βοηθητική συνάρτηση βρίσκεται στο `/usr/src/lib/posix/_fork.c` εφόσον πρόκειται για μια λειτουργία που ορίζεται από το πρότυπο POSIX



Βοηθητική Συνάρτηση για την Κλήση Συστήματος fork (2)

```
#include <lib.h>
#define fork _fork
#include <unistd.h>

PUBLIC pid_t fork() {
    message m;
    return(_syscall(MM, FORK, &m));
}
```

- ▶ Απλή επικοινωνία – δεν χρειάζεται κάποιο όρισμα
- ▶ Χρησιμοποιούμε την βασική δομή μηνυμάτων
- ▶ Το αποτέλεσμα που επιστρέφει η κλήση είναι η μοναδική πληροφορία που επιστρέφει η συνάρτηση
- ▶ Η τιμή της FORK ορίζεται στο *include/minix/callnr.h*
- ▶ Η τιμή της MM ορίζεται στο *include/lib.h* – είναι ο διαχειριστής διεργασιών (process manager)



Χρήση Κλήσεων Συστήματος

- ▶ Η βοηθητική συνάρτηση δεν στέλνει το μήνυμα – χρησιμοποιεί μια δεύτερη βοηθητική συνάρτηση (*_syscall*)
- ▶ Η συνάρτηση *_syscall* είναι αυτή που αναλαμβάνει να στείλει το μήνυμα
- ▶ Πρόκειται για μια συνάρτηση που ορίζεται στο αρχείο */usr/src/lib/other/syscall.c*
 - ▶ Γενικά – οι συναρτήσεις που προσφέρει το Λ.Σ. για την υλοποίηση εντολών/εφαρμογών βρίσκονται στον φάκελο */usr/src/lib*
 - ▶ ... λέμε ότι ανήκουν στις βιβλιοθήκες του Λ.Σ.
- ▶ Επίσης η συνάρτηση *_syscall* περιμένει να ολοκληρωθεί η κλήση στο σύστημα
 - ▶ Παραλαμβάνει το μήνυμα που περιέχει την απάντηση
 - ▶ Κάνει Send – Receive



Βοηθητική Συνάρτηση για εκτέλεση των Κλήσεων Συστήματος

```
#include <lib.h>

PUBLIC int _syscall(int who, int syscallnr,
                    message *msgptr) {

    int status;
    msgptr->m_type = syscallnr;
    status = _sendrec(who, msgptr);
    if (status != 0) {
        /* 'sendrec' itself failed. */
        msgptr->m_type = status;
    }
    if (msgptr->m_type < 0) {
        errno = -msgptr->m_type;
        return(-1);
    }
    return(msgptr->m_type);
}
```



Αρχή Rendezvous

- ▶ Η επικοινωνία μεταξύ των διεργασιών γίνεται μέσω ανταλλαγής μηνυμάτων
- ▶ Εφαρμόζεται η αρχή "rendezvous"
- ▶ Όταν μια διεργασία καλεί την *send* τα χαμηλότερα επίπεδα του πυρήνα ελέγχουν κατά πόσο ο παραλήπτης είναι σε αναμονή
 - ▶ **Αν είναι**, το μήνυμα αντιγράφεται στην περιοχή της μνήμης του παραλήπτη – οι δύο διεργασίες μαρκάρονται ως "runnable"
 - ▶ **Αν δεν είναι** – ο αποστολέας μαρκάρεται ως "blocked" και τοποθετείται σε μια ουρά αναμονής
- ▶ Αντίστοιχα, όταν μια διεργασία καλεί την *receive* ο πυρήνας ελέγχει κατά πόσο ο αποστολέας είναι έτοιμος να στείλει
- ▶ Με αυτόν τον τρόπο αποφεύγεται η υλοποίηση ενδιάμεσων buffers για την αποθήκευση των μηνυμάτων που έχουν αποσταλεί αλλά δεν έχουν παραληφθεί ακόμα



Αποστολή/Παραλαβή

- ▶ Όταν μια διεργασία θέλει να κάνει κλήση στο σύστημα, στέλνει ένα μήνυμα
- ▶ Ο πυρήνας φροντίζει ο κατάλληλος διαχειριστής/οδηγός να παραλάβει το μήνυμα
- ▶ Ο διαχειριστής ολοκληρώνει την κλήση και αμέσως μετά στέλνει μια απάντηση στην διεργασία (το αποτέλεσμα της κλήσης)
 - ▶ Τι θα γίνει αν η διεργασία δεν περιμένει για απάντηση ;
 - ▶ Είναι δυνατόν ο διαχειριστής/οδηγός να κάνει "block" ;
- ▶ Για αυτόν τον λόγο, οι κλήσεις στο σύστημα γίνονται με την συνάρτηση `_sendrc()` – Αποστολή/Παραλαβή
 - ▶ Στέλνει το μήνυμα και αμέσως μετά περιμένει για τη απάντηση
 - ▶ Ο διαχειριστής/οδηγός δεν θα κάνει "block"
- ▶ Για αντίστοιχο λόγο χρησιμοποιούμε την `_sendrc()` για τις κλήσεις του πυρήνα



Αδιέξοδα στις Κλήσεις Συστήματος / Πυρήνα

- ▶ Μια διεργασία A θέλει να στείλει στην B (ή να στείλει και να περιμένει απάντηση)
- ▶ Η διεργασία B μπορεί να θέλει να στείλει και αυτή μήνυμα στην A
 - ▶ Αυτό μπορεί να οδηγήσει σε αδιέξοδο
- ▶ Αν πρόκειται για κάποια κλήση του συστήματος, τα μηνύματα στέλνονται από την `sys_call`
 - ▶ Φροντίζει για την αποφυγή της παραπάνω περίπτωσης
- ▶ Για κάθε διεργασία, αντιστοιχεί μια λίστα από διεργασίες που είναι έτοιμες να σταλούν και έχουν γίνει "blocked" (περιμένουσ)
- ▶ Αρχικά ελέγχει απλά κατά πόσο η λίστα είναι άδεια
- ▶ Στην συνέχεια, ελέγχει τις διεργασίες που είναι στην λίστα
- ▶ Με αυτόν τον τρόπο μπορεί να ελέγξει "απλούς κύκλους"



Αποστολή – Παραλαβή Μηνυμάτων

- ▶ Η υλοποίηση της `Send – Receive` γίνεται από την συνάρτηση `_sendrc()`
- ▶ Αναλαμβάνει να στείλει το μήνυμα στην διεργασία
- ▶ Ορίζεται στο αρχείο `src/include/minix/ipc.h`

```
#define sendrc _sendrc
_PROTOTYPE(int sendrc, (int src_dest, message *m_ptr))
```

- ▶ Υλοποιείτε σε επίπεδο assembly στο αρχείο `src/lib/i386/rtc/_sendrc.s`
- ▶ Βασίζεται στην συνάρτηση `_sendrc` – επίσης υλοποιείτε σε επίπεδο assembly στο αρχείο `src/lib/i386/rtc/_ipc.s`
- ▶ **Αντιγράφει το μήνυμα στην διεργασία – παραλήπτης και αμέσως μετά στέλνει ένα interrupt SYSVEC στον πυρήνα**
- ▶ Η εκτέλεση της διεργασίας – αποστολέας διακόπεται και ενεργοποιείται η διεργασία – παραλήπτης



Διαχείριση Interrupt

- ▶ Ο πυρήνας είναι υπεύθυνος για την διαχείριση των interrupt
 - ▶ Ήθε πρόκειται για hardware interrupt ή για software interrupt
- ▶ Μόλις δεχθεί το interrupt SYSVEC – εκτελεί τον αντίστοιχο interrupt handler
- ▶ Εφόσον πρόκειται για software interrupt περνάει την διαχείριση στην συνάρτηση `_s_call`
 - ▶ Ορίζεται στο αρχείο `/usr/src/kernel/mpx386.s`
 - ▶ Υλοποιείται σε επίπεδο assembly -- το χαμηλότερο επίπεδο του πυρήνα
 - ▶ Αντιγράφει τα στοιχεία του μηνύματος και περνάει τον έλεγχο στην συνάρτηση `sys_call`
- ▶ Η συνάρτηση `sys_call` ορίζεται στο αρχείο `/usr/src/kernel/proc.c`
 - ▶ Ελέγχει τα στοιχεία του μηνύματος (αποστολέας, παραλήπτης, τύπος, δικαιώματα κλπ.)
 - ▶ Παραδίδει το μήνυμα στον παραλήπτη – διαχειριστή/οδηγό



Συνάρτηση sys_call (1)

- ▶ Η συνάρτηση `sys_call` ορίζεται στο αρχείο `/usr/src/kernel/proc.c`
- ▶ Η ολοκλήρωση της αποστολής γίνεται από την συνάρτηση `mini_send`
- ▶ Ο πίνακας των διεργασιών έχει το πεδίο `bit p_rts_flag` – ενεργοποιείται όταν η διεργασία έχει γίνει block λόγω κάποιου `receive`
 - ▶ Αν όντως περιμένει – η επόμενη ερώτηση είναι 'ποιόν περιμένει';
 - ▶ Αν περιμένει τον αποστολέα (ή οποιοδήποτε – ANY) το μήνυμα αντιγράφεται και η επικοινωνία ολοκληρώνεται

```
/* Check if 'dst' is blocked waiting for this message. The destination's
 * SENDING flag may be set when its SENDREC call blocked while sending.
 */
if ((dst_ptr->p_rts_flags & (RECEIVING | SENDING)) == RECEIVING &&
    (dst_ptr->p_getfrom_e == ANY
     || dst_ptr->p_getfrom_e == caller_ptr->p_endpoint)) {
    /* Destination is indeed waiting for this message. */
    CopyMsg(caller_ptr->p_nr, caller_ptr, m_ptr, dst_ptr,
            dst_ptr->p_messbuf);
}
```



Συνάρτηση sys_call (2)

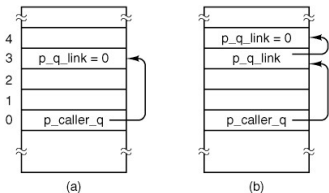
- ▶ Αν η διεργασία - παραλήπτης δεν είναι σε αναμονή (δεν κάνει `receive` – ή περιμένει μήνυμα από άλλη διεργασία
 - ▶ Η διεργασία - αποστολέας γίνεται block
- ▶ Όλες οι διεργασίες που έχουν γίνει block περιμένοντας μια συγκεκριμένη διεργασία, δημιουργούν μια λίστα (ουρά αναμονής)
 - ▶ Χρησιμοποιούμε το πεδίο `p_callerq` για να 'δείξουμε' την εγγραφή στον πίνακα των διεργασιών που είναι 'μπροστά' στην λίστα (ουρά)

```
} else if (! (flags & NON_BLOCKING)) {
    /* Destination is not waiting. Block and dequeue caller. */
    caller_ptr->p_messbuf = m_ptr;
    if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
    caller_ptr->p_rts_flags |= SENDING;
    caller_ptr->p_sendto_e = dst_e;

    /* Process is now blocked. Put in on the destination's queue. */
    xpp = &dst_ptr->p_caller_q; /* find end of list */
    while (*xpp != NIL_PROC) xpp = &(*xpp)->p_q_link;
    *xpp = caller_ptr; /* add caller to end */
    caller_ptr->p_q_link = NIL_PROC; /* mark new end of list */
}
```



Συνάρτηση sys_call (3)



- (a) Η διεργασία 3 δεν μπορεί να στείλει στην διεργασία 0
- (b) Στην συνέχεια, η διεργασία 4 επικειρεί και αυτή (ανεπιτυχώς) να στείλει στην διεργασία 0



Διαχειριστής Διεργασιών

- ▶ Η συνάρτηση `sys_call` παραδίδει το μήνυμα στον διαχειριστή διεργασιών
- ▶ Η συνάρτηση `get_work` παραλαμβάνει το μήνυμα
 - ▶ Εκτελεί ορισμένους βασικούς ελέγχους
- ▶ Ο διαχειριστής διεργασιών ανατρέχει στον πίνακα `call_vec`
 - ▶ Εντοπίζει την συνάρτηση που είναι υπεύθυνη για την διαχείριση της συγκεκριμένης κλήσης
- ▶ Εφόσον πρόκειται για την κλήση FORK χρησιμοποιεί την συνάρτηση `do_fork` που ορίζεται στο αρχείο `/usr/src/servers/pm/forkexit.c`
 - ▶ Ελέγχει κατά πόσο μπορεί να δημιουργηθεί μια νέα διεργασία
 - ▶ Αποφασίζει το μέγεθος της μνήμης που θα αναθέσει
 - ▶ Αντιγράφει το `image` της διεργασίας-γονέας στην νέα διεργασία
 - ▶ Ενημερώνει τον πίνακα διεργασιών
- ▶ Καλεί την κλήση του πυρήνα SYS_FORK



Κλήση Πυρήνα από τον Διαχειριστή Διεργασιών

```
PUBLIC int do_fork() {
    ...
    if((r=sys_fork(who_e, child_nr, &rmc->mp_endpoint))!=OK)
        panic(__FILE__, "do_fork can't sys_fork", r);
}
...
}
```

- ▶ Η συνάρτηση `sys_fork` είναι μια βοηθητική συνάρτηση
 - ▶ Όπως και στις κλήσεις του συστήματος (system calls) την χρησιμοποιούμε για λόγους ευκολίας
 - ▶ 'κρύβει' τους μηχανισμούς ανταλλαγής μηνυμάτων
 - ▶ Η βοηθητική συνάρτηση δημιουργεί ένα νέο μήνυμα (που περιέχει τις παραμέτρους της κλήσης του πυρήνα) και στέλνει το μήνυμα στο system task
 - ▶ Συγκεκριμένα για την `fork0` η βοηθητική συνάρτηση βρίσκεται στο `/usr/src/lib/syslib/sys_fork.c`



Βοηθητική Συνάρτηση για την Κλήση Πυρήνα fork

```
#include "syslib.h"

PUBLIC int sys_fork(parent, child, child_endpoint)
int parent; /* process doing the fork */
int child; /* which proc has been created by the fork */
int *child_endpoint;
{
    /* A process has forked. Tell the kernel. */
    message m;
    int r;

    m.PR_ENDPT = parent;
    m.PR_SLOT = child;
    r = _taskcall(SYSTASK, SYS_FORK, &m);
    *child_endpoint = m.PR_ENDPT;
    return r;
}
```



Χρήση Κλήσεων Πυρήνα

- ▶ Η βοηθητική συνάρτηση δεν στέλνει το μήνυμα – χρησιμοποιεί μια δευτερεύουσα βοηθητική συνάρτηση (`_taskcall`)
 - ▶ Παρόμοια με την `_syscall`
 - ▶ αναλαμβάνει να στείλει το μήνυμα
- ▶ Η συνάρτηση ορίζεται στο αρχείο `/usr/src/lib/other/taskcall.c`
- ▶ Όπως και με την συνάρτηση `_syscall`, η συνάρτηση `_taskcall` περιμένει να ολοκληρωθεί η κλήση στο σύστημα
 - ▶ Παραλαμβάνει το μήνυμα που περιέχει την απάντηση
 - ▶ Κάνει Send – Receive
- ▶ Η βασική διαφορά είναι ότι δεν επιστρέφει κάποια τιμή που είναι το αποτέλεσμα της κλήσεως, όπως κάνει η `_syscall`
 - ▶ Αν συμβεί λάθος επιστρέφει το `errno`
 - ▶ Για διάφορους λόγους, αυτό διευκολύνει τους διαχειριστές



Βοηθητική Συνάρτηση για εκτέλεση των Κλήσεων Πυρήνα

```
#include <lib.h>
#include <minix/syslib.h>

PUBLIC int _syscall(int who, int syscallnr,
                    message *msgptr) {
    int status;
    msgptr->m_type = syscallnr;
    status = _sendrec(who, msgptr);
    if (status != 0) return(status);
    return(msgptr->m_type);
}
```

- ▶ Η Send – Receive γίνεται από την συνάρτηση `_sendrec0`
- ▶ Αντιγράφει το μήνυμα και στέλνει ένα interrupt `SYSVEC`
- ▶ Η εκτέλεση του διαχειριστή διακόπτεται και αναλαμβάνει ο πυρήνας



Σύνοψη 7ης Διάλεξης

Προηγούμενο Μάθημα

To System Task

Οργάνωση του System Task

Υλοποίηση του System Task

Λειτουργικό Σύστημα Minix

Διαχειριστής Διεργασιών – Κλήσεις Συστήματος

Ροή Εκτέλεσης Κλήσης Συστήματος

Θέματα Σχεδιασμού

Σύνοψη Μαθήματος

Σύνοψη Μαθήματος

Δεύτερη Άσκηση

Επόμενη Διάλεξη



Στο σημερινό μάθημα είδαμε

- ▶ Το System Task στο Minix 3
- ▶ Ροή Εκτέλεσης Κλήσεων Συστήματος
 - ▶ Διασύνδεση με Κλήσεις Πυρήνα
- ▶ Επικοινωνία Διεργασιών
- ▶ Υλοποίηση Κλήσεων Συστήματος



Βιβλιογραφία

- ▶ Βιβλίο "Operating Systems Design and Implementation, Third Edition" (Andrew S. Tanenbaum)
 1. Κεφάλαιο 2: Processes
 - ▶ Παράγραφος 2.6.9 Interprocess Communication in MINIX 3
 - ▶ Παράγραφος 2.7 The System Task in Minix 3
 2. Κεφάλαιο 4: Memory Management
 - ▶ Παράγραφος 4.7.4 The FORK, EXIT, and WAIT System Calls



Δεύτερη Άσκηση

- ▶ Αφορά το λειτουργικό σύστημα MINIX 3
- ▶ Οι απαντήσεις πρέπει να ακολουθούν συγκεκριμένη μορφή
- ▶ Ατομική άσκηση
- ▶ Παράδοση γίνεται με την χρήση του εργαλείου *submit*
- ▶ Μέχρι **Τετάρτη 13 Μαΐου, ώρα 23:59**
- ▶ Σε περίπτωση που η άσκηση παραδοθεί με καθυστέρηση, για κάθε εβδομάδα καθυστέρησης θα υπάρξει μείωση 30%
- ▶ Αν παρατηρηθεί αντιγραφική, τότε όλες οι ομάδες που συνεργάστηκαν και εμπλέκονται στην αντιγραφική, θα μηδενίζονται στο μάθημα



1^ο Πρόβλημα

Περιγράψτε τη διαδικασία που πρέπει να ακολουθήσουμε για να δημιουργήσουμε ένα νέο image του minix όπου η κλήση πυρήνα SYS_NICE να είναι απενεργοποιημένη.



2^ο Πρόβλημα

- ▶ Αναπτύξτε μια νέα κλήση του συστήματος (system call) στα πλαίσια του διαχειριστή διεργασιών (process manager)
- ▶ Δέχεται έναν ακέραιο ως παράμετρο
- ▶ Επιστρέφει το τετράγωνο του ακεραίου.
- ▶ Υλοποιείστε μια απλή εντολή που χρησιμοποιεί την νέα κλήση του συστήματος που αναπτύξατε.



3^ο Πρόβλημα

- ▶ Αναπτύξτε μια νέα κλήση του πυρήνα (kernel call)
- ▶ Δέχεται έναν ακέραιο ως παράμετρο
- ▶ Επιστρέφει το διπλάσιο του ακεραίου.
- ▶ Υλοποιείστε μια απλή εντολή που χρησιμοποιεί την νέα κλήση του πυρήνα που αναπτύξατε.



Επόμενη Διάλεξη

- ▶ Διεργασίες στο Λ.Σ. MINIX 3
- ▶ Διαχείριση Μνήμης στο Λ.Σ. MINIX 3
- ▶ Επανάληψη από μάθημα 'Λειτουργικά Συστήματα Ι'
 1. Κεφάλαιο 2: Διεργασίες
 2. Κεφάλαιο 4: Διαχείριση Μνήμης
- ▶ Βιβλίο 'Σύγχρονα Λειτουργικά Συστήματα' (A.Tanenbaum)
 1. Κεφάλαιο 2: Διεργασίες
 2. Κεφάλαιο 4: Διαχείριση Μνήμης
 3. Κεφάλαιο 10: Μελέτη Περίπτωσης 1: Unix και Linux
 - ▶ Παράγραφος 10.4 Η Διαχειριστή Μνήμης στο UNIX
- ▶ Βιβλίο 'Operating Systems Design and Implementation, Third Edition'' (Andrew S. Tanenbaum)
 1. Κεφάλαιο 2: Processes
 2. Κεφάλαιο 4: Memory Management

