Εργαστήριο Λειτουργικών Συστημάτων

Μάθημα 6^{ου} Εξαμήνου,
Τομέας Λογικού και Υπολογιστών

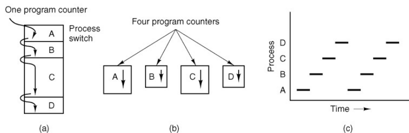
Ιωάννης Χατζηγιαννάκης

Σημειώσεις Μαθήματος
Ενότητα 4

Σημειώσεις Μαθήματος – 4^{η} Ενότητα

Διεργασίες στο MINIX 3
- Διεργασίες
- Χρονοπρογραμματισμός στα αλληλεπιδραστικά συστήματα
- Υλοποίηση Μηχανισμού Χρονοπρογραμματισμού
- Διαχείριση Μνήμης
- Διαχείριση Ελεύθερης Μνήμης
- Υλοποίηση Διαχείρισης Μνήμης
- Σήματα
- Λειτουργίες Σημάτων
- Υλοποίηση Διαχείρισης Σημάτων
- Ανταλλαγή Δεδομένων
- Αποστολή Σημάτων
- Επιστροφή απο Χειριστή Σημάτων

Εκτέλεση Διεργασιών



- Οι διεργασίες εκτελούνται σειριακά ανά διαθέσιμο επεξεργαστή
- Μόνο μια διεργασία είναι ενεργή ανά πάσα στιγμή
- Οι πληροφορίες για κάθε διεργασία βρίσκονται: στο επίπεδο του πυρήνα, στον διαχειριστή διεργασιών και στον διαχειριστή αρχείων
- Το Λ.Σ. χρονοπρογραμματίζει τις διεργασίες – στο επίπεδο του πυρήνα και στον διαχειριστή διεργασιών

Κύκλος Ζωής Διεργασιών

- Τέσσερα βασικά γεγονότα οδηγούν στην δημιουργία νέων διεργασιών
  1. Αρχικοποίηση του συστήματος
  2. Μια διεργασία εκτελεί την κλήση συστήματος για την δημιουργία νέας διεργασίας
  3. Ένας χειριστής εκτελεί μια εντολή
  4. Εκκίνηση μιας εργασίας (batch job)
- Οι διεργασίες τερματίζουν εάν προκύψει ένας από τους παρακάτω λόγους
  1. Κανονικός τερματισμός – έξοδος
  2. Τερματισμός λόγο λάθους
  3. Απότομος τερματισμός
  4. Τερματισμός από μια άλλη διεργασία

Ιεράρχηση Διεργασιών

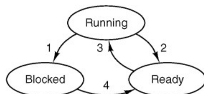
- Όταν μια διεργασία δημιουργήσει μια νέα διεργασία (μέσω κάποιας κλήσης στο σύστημα)
  1. Θεωρούμε ότι υπάρχει σχέση γονέα – παιδιού
- Μια διεργασία παιδί μπορεί να δημιουργήσει και αυτή νέες διεργασίες
- Δημιουργείτε μαι ιεραρχία διεργασιών – ένα δένδρο
- Οι διεργασίες μπορούν να επικοινωνήσουν με τα κατώτερα επίπεδα του Λ.Σ. είτε αυτόνομα είτε ως ιεραρχία διεργασιών (ομάδα)
  1. π.χ. ένα σήμα παραδίδεται σε όλες τις διεργασίες
- Το Minix χρησιμοποιεί δύο 'ειδικές' διεργασίες που δημιουργούντε κατά την αρχικοποίησης του συστήματος
  - Reincarnation Server και init
  - δημιουργούν όλες τις άλλες διεργασίες του συστήματος

Reincarnation Server και init

- Το init είναι η πρώτη διεργασία που εκτελείτε κατά την αρχικοποίηση του συστήματος
- Εκτελεί τα scripts που βρίσκονται στον φάκελο */etc/rc* μέσω του reincarnation server
- Η βασική λειτουργία του reincarnation server (rs) είναι η εκκίνηση των servers και των οδηγών (device drivers)
- Επομένως όλοι οι servers και οι οδηγοί είναι 'παιδιά' του rs
- Αν κάποια διεργασία τερματίσει τότε θα ειδοποιηθεί ο rs
- Παρακολουθεί την εκτέλεση του και αν διαπιστώσει κάποιο πρόβλημα έχει το δικαίωμα να επανεκκινησει την διεργασία
- Με αυτόν τον τρόπο το Minix προσφέρει λειτουργίες αυτο-ίασης
- Η τελευταία διεργασία που ξεκινάει η init είναι η *getty* που είναι υπεύθυνη για τις κονσόλες των χειριστών

Εκτέλεση Διεργασιών

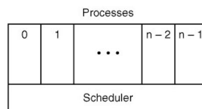
- Οι διεργασίες είναι ανεξάρτητες οντότητες
- Μπορούν να αλληλεπιδράσουν μεταξύ τους, να επικοινωνήσουν και να συγχρονιστούν
- `cat chapter1 chapter2 | grep tree`
  - Η grep είναι έτοιμη να 'τρέξει' όμως περιμένει είσοδο από την cat
- Επομένως μια διεργασία μπλοκάρει επειδή δεν μπορεί να συνεχίσει την εκτέλεση της
- Υπάρχουν τέσσερις πιθανοί τρόποι μετάβασης μεταξύ των καταστάσεων μιας διεργασίας:



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Υλοποίηση Διεργασιών

- Η υλοποίηση του μοντέλου των διεργασιών βασίζεται στην διατήρηση του 'πίνακα στοιχείων διεργασιών' (process table)
  - Περιέχει μια εγγραφή για κάθε διεργασία
  - Κάθε εγγραφή



- Η διαχείριση των διεργασιών δεν γίνεται σε ένα σημείο
  - Ορισμένες λειτουργίες διαχείρισης γίνεται στο επίπεδο του πυρήνα
  - Κάποιες άλλες λειτουργίες γίνονται από τον διαχειριστή διεργασιών και τον διαχειριστή αρχείων
- Άρα δεν υπάρχει ένας πίνακας αλλά τρεις!

Πίνακες Στοιχείων Διεργασιών

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Time when process started CPU time used Children's CPU time Time of next alarm Message queue pointers Pending signal bits Process id Various flag bits	Pointer to text segment Pointer to data segment Pointer to bss segment Exit status Signal status Process id Parent process Process group Real uid Effective uid Real gid Effective gid Bit maps for signals Various flag bits	UMASK mask Root directory Working directory File descriptors Effective uid Effective gid System call parameters Various flag bits

Πίνακας Διεργασιών – Σταθερές

- Στο αρχείο */usr/src/include/minix/com.h* ορίζονται κάποιες σταθερές για όλο το σύστημα

```
// Number of tasks
#define NR_TASKS    4
```

- Στο αρχείο */usr/src/include/minix/sys_config.h* ορίζονται κάποιες σταθερές για όλο το σύστημα

```
// Number of processes
#define _NR_PROCS 100
// Number of system processes
#define _NR_SYS_PROCS 32
```

Πίνακας Διεργασιών του Διαχειριστή Διεργασιών (1)

```
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 */
  ...
```

Πίνακας Διεργασιών του Διαχειριστή Διεργασιών (2)

```
  ...
  /* Signal handling information. */
  sigset_t mp_ignore; /* 1 means ignore the signal, 0 me
  sigset_t mp_catch; /* 1 means catch the signal, 0 mean
  sigset_t mp_sig2mess; /* 1 means transform into notify
  sigset_t mp_sigmask; /* signals to be blocked */
  sigset_t mp_sigmask2; /* saved copy of mp_sigmask */
  sigset_t mp_sigpending; /* pending signals to be handl
  struct sigaction mp_sigact[_NSIG + 1]; /* as in sigact
  vir_bytes mp_sigreturn;  /* address of C library __sig
  struct timer mp_timer; /* watchdog timer for alarm(2)

  /* Scheduling priority. */
  signed int mp_nice; /* nice is PRIO_MIN..PRIO_MAX, sta
  char mp_name[PROC_NAME_LEN]; /* process name */
} mproc[NR_PROCS];
```

Πίνακας Διεργασιών του Συστήματος Αρχείων (1)

```
EXTERN struct fproc {
  mode_t fp_umask; /* mask set by umask system call */
  struct inode *fp_workdir; /* pointer to working direct
  struct inode *fp_rootdir; /* pointer to current root d
  struct filp *fp_filp[OPEN_MAX];/* the file descriptor
  fd_set fp_filp_inuse; /* which fd's are in use? */
  uid_t fp_realuid; /* real user id */
  uid_t fp_effuid; /* effective user id */
  gid_t fp_realgid; /* real group id */
  gid_t fp_effgid; /* effective group id */
  dev_t fp_tty; /* major/minor of controlling tty */
  int fp_fd; /* place to save fd if rd/wr can't finish *
  ...
```

Πίνακας Διεργασιών του Συστήματος Αρχείων (2)

```
  ...
  char *fp_buffer; /* place to save buffer if rd/wr can'
  int  fp_nbytes; /* place to save bytes if rd/wr can't
  int  fp_cum_io_partial; /* partial byte count if rd/wr
  char fp_suspended; /* set to indicate process hanging
  char fp_revived; /* set to indicate process being revi
  int fp_task; /* which task is proc suspended on */
  char fp_sesldr; /* true if proc is a session leader */
  char fp_execced; /* true if proc has exec()ced after f
  pid_t fp_pid; /* process id */
  long fp_cloexec; /* bit map for POSIX Table 6-2 FD_CLO
  int fp_endpoint; /* kernel endpoint number of this pro
} fproc[NR_PROCS];
```

Δομή Πίνακα Διεργασιών του Πυρήνα (1)

- Στο αρχείο */usr/src/kernel/proc.h* ορίζεται η δομή του πίνακα των διεργασιών (για τον πυρήνα)

```
struct proc {
  // process' registers saved in stack frame
  struct stackframe_s p_reg;
  // number of this process (for fast access)
  proc_nr_t p_nr;
  // system privileges structure
  struct priv *p_priv;
  // process is runnable only if zero
  short p_rts_flags;
  // flags that do suspend the process
  short p_misc_flags;
```

Δομή Πίνακα Διεργασιών του Πυρήνα (2)

```
  // current scheduling priority
  char p_priority;
  // maximum scheduling priority
  char p_max_priority;
  // number of scheduling ticks left
  char p_ticks_left;
  // quantum size in ticks
  char p_quantum_size;

  // memory map (T, D, S)
  struct mem_map p_memmap[NR_LOCAL_SEGS];

  // user/sys time in ticks
  clock_t p_user_time;
  clock_t p_sys_time;
```

Δομή Πίνακα Διεργασιών του Πυρήνα (3)

```
// pointer to next ready process
struct proc *p_nextready;
// head of list of procs wishing to send
struct proc *p_caller_q;
// link to next proc wishing to send
struct proc *p_q_link;
// pointer to passed message buffer
message *p_messbuf;
// from whom does process want to receive?
int p_getfrom_e;
// to whom does process want to send?
int p_sendto_e;
// bit map for pending kernel signals
sigset_t p_pending;
```

Δομή Πίνακα Διεργασιών του Πυρήνα (4)

```
  // name of the process, including \0
  char p_name[P_NAME_LEN];

  // endpoint number, generation-aware
  int p_endpoint;

  // Debug info for scheduler
  int p_ready, p_found;
}
```

Αδιέξοδα στις Κλήσεις Συστήματος / Πυρήνα

- Μια διεργασία Α θέλει να στείλει στην Β (ή να στείλει και να περιμένει απάντηση)
- Η διεργασία Β μπορεί να θέλει να στείλει και αυτή μήνυμα στην Α
  - 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 στείλουν και έχουν γίνει ``blocked'' (περιμένουν)
- 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'

Διαχείριση Interrupt

- Κάθε φορά που στέλνουμε ένα μήνυμα δημιουργείτε ένα interrupt
- Ο πυρήνας είναι υπεύθυνος για την διαχείριση των interrupt
  - Μόλις δεχθεί το interrupt SYSVEC -- εκτελεί τον αντίστοιχο interrupt handler
- Δίνει τον έλεγχο και 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. */
      CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, dst_ptr,
               dst_ptr->p_messbuf);
```

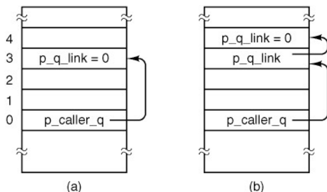
Συνάρτηση *sys_call* (2)

- Αν η διεργασία - παραλήπτης δεν είναι σε αναμονή (δεν κάνει receive -- ή περιμένει μήνυμα από άλλη διεργασία
  - Η διεργασία - αποστολέας γίνεται block
- 'Ολες 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

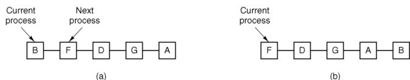
(b) Στην συνέχεια, η διεργασία 4 επιχειρεί και αυτή (ανεπιτυχώς) να στείλει στην διεργασία 0

Χρονοπρογραμματισμός διεργασιών

- Κάθε φορά που προκύπτει ένα interrupt -- είναι μια ευκαιρία να επανεκτιμήσει το Λ.Σ. ποια διεργασία δικαιούται να χρησιμοποιήσει τον επεξεργαστή (να ενεργοποιηθεί)
- Το Minix 3 χρησιμοποιεί ένα σύστημα ουρών πολλαπλών επιπέδων
  - Ορίζει 16 ουρές – κάθε ουρά αντιπροσωπεύει ένα επίπεδο προτεραιότητας
  - Οι διεργασίες που είναι IDLE βρίσκονται στο χαμηλότερο επίπεδο
  - Το System Task και το ρολόι εκτελούνται στο υψηλότερο επίπεδο
- Εκτός από την προτεραιότητα που ορίζεται από την ουρά, χρησιμοποιούμε και τον μηχανισμό quantum
  - Κάθε διεργασία δικαιούται να χρησιμοποιήσει τον επεξεργαστή μόνο για ένα συγκεκριμένο χρονικό διάστημα – quantum
  - 'Οταν ολοκληρωθεί το quantum γίνεται block έως ότου ξανά έρθει η σειρά της
  - Το μέγεθος του quantum είναι διαφορετικό για κάθε κατηγορία διεργασιών

Χρονοπρογραμματισμός εκ περιτροπής

- Κάθε διεργασία δικαιούται να χρησιμοποιήσει τον επεξεργαστή μόνο για ένα συγκεκριμένο χρονικό διάστημα – quantum
  - Όταν ολοκληρωθεί το quantum (κβάντο χρόνου) γίνεται block έως ότου ξανά έρθει η σειρά της
  - Τοποθετείτε στο τέλος της ουράς

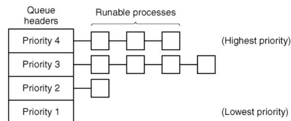


(a) Η λίστα των εκτελέσιμων διεργασιών

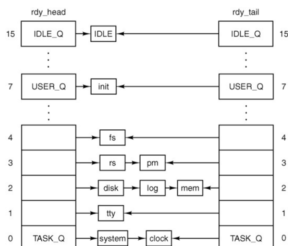
(b) Η λίστα των εκτελέσιμων διεργασιών μετά το πέρας του κβάντου (quantum) της διεργασίας B

Χρονοπρογραμματισμός με βάση την προτεραιότητα

- Όλες οι διεργασίες δεν είναι εξίσου σημαντικές
  - Σε κάθε διεργασία αντιστοιχίζεται μια προτεραιότητα
- Σε κάθε χρονική στιγμή, εκτελείται η (έτοιμη) διεργασία με την υψηλότερη προτεραιότητα (priority scheduling)
- Σε κάθε διεργασία αντιστοιχίζεται ένα μέγιστο κβάντο (quantum)
  - Καθορίζει πόσο χρόνο επιτρέπεται να εκτελείται κάθε φορά
- Οι προτεραιότητες μπορούν να ανατεθούν στατικά ή δυναμικά



Χρονοπρογραμματισμός στο Minix 3



- Βασίζεται στην προηγούμενη τεχνική
- Σε κάθε ουρά γίνεται εκ περιτροπής
- Ορίζει 16 ουρές
- Θέτει διαφορετικό quantum για κάθε διεργασία
- Μπορούν εύκολα να αλλάξουν
- Γίνεται από τον πυρήνα

Εσωτερική Δομή Πυρήνα – Σταθερές

- Στο αρχείο */usr/src/kernel/proc.h* ορίζονται οι σταθερές (για τον πυρήνα) που αφορούν τα θέματα χρονοπρογραμματισμού

Process scheduling constants

```
// MUST equal minimum priority + 1
#define NR_SCHED_QUEUES   16
// highest, used for kernel tasks
#define TASK_Q            0
// default (should correspond to nice 0)
#define USER_Q            7
// lowest, only IDLE process goes here
#define IDLE_Q            15
// highest/loweste priority for user processes
#define MAX_USER_Q        0
#define MIN_USER_Q        14
```

Εσωτερική Δομή Πυρήνα – Καθολικές Μεταβλητές (1)

- Στο αρχείο */usr/src/kernel/proc.h* ορίζονται οι καθολικές μεταβλητές (για τον πυρήνα) για τον πίνακα των διεργασιών και κάποιοι βοηθητικοί πίνακες

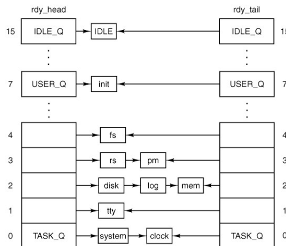
```
// process table
EXTERN struct proc proc[NR_TASKS + NR_PROCS];

// pointers to process table slots
EXTERN struct proc *pproc_addr[NR_TASKS+NR_PROCS

// ptrs to ready list headers
EXTERN struct proc *rdy_head[NR_SCHED_QUEUES];

// ptrs to ready list tails
EXTERN struct proc *rdy_tail[NR_SCHED_QUEUES];
```

Εσωτερική Δομή Πυρήνα – Καθολικές Μεταβλητές (2)



- Η *rdy_head* έχει 1 εγγραφή για κάθε ουρά
- Κάθε εγγραφή είναι δείκτης (pointer) στην διεργασία που είναι στην αρχή της ουράς
- Η *rdy_tail* είναι ένας αντίστοιχος πίνακας
- Δείχνει στην διεργασία που είναι στο τέλος της ουράς

Εσωτερική Δομή Πυρήνα – Καθολικές Μεταβλητές (3)

- Στο αρχείο */usr/src/kernel/glo.h* ορίζονται οι καθολικές μεταβλητές (για τον πυρήνα)
- Είναι αναφορές (pointers) στον πίνακα των διεργασιών που διατηρεί ο πυρήνας

```
/* previously running process */
EXTERN struct proc *prev_ptr;
/* pointer to currently running process */
EXTERN struct proc *proc_ptr;
/* next process to run after restart() */
EXTERN struct proc *next_ptr;
/* process to bill for clock ticks */
EXTERN struct proc *bill_ptr;
```

Εσωτερική Δομή Πυρήνα – Αρχικοποίηση Δομών

- Στο αρχείο */usr/src/kernel/table.c* ορίζονται οι αρχικές διεργασίες και η προτεραιότητα και το κβάντο που θα έχουν

```
PUBLIC struct boot_image image[] = {
/* process nr,    pc, flags, qs,  queue, stack, traps, ipcto, call,  name */
 { IDLE,  idle_task, IDL_F,  8, IDLE_Q, IDL_S,     0,     0,     0, "idle"  },
 { CLOCK,clock_task, TSK_F,  8, TASK_Q, TSK_S, TSK_T,     0,     0, "clock" },
 { SYSTEM, sys_task, TSK_F,  8, TASK_Q, TSK_S, TSK_T,     0,     0, "system"},
 { HARDWARE,      0, TSK_F,  8, TASK_Q, HRD_S,     0,     0,     0, "kernel"},
 { PM_PROC_NR,    0, SRV_F, 32,      3, 0,     SRV_T, SRV_M,  PM_C, "pm"    },
 { FS_PROC_NR,    0, SRV_F, 32,      4, 0,     SRV_T, SRV_M,  FS_C, "fs"    },
 { RS_PROC_NR,    0, SRV_F,  4,      3, 0,     SRV_T, SYS_M,  RS_C, "rs"    },
 { DS_PROC_NR,    0, SRV_F,  4,      3, 0,     SRV_T, SYS_M,  DS_C, "ds"    },
 { TTY_PROC_NR,   0, SRV_F,  4,      1, 0,     SRV_T, SYS_M, TTY_C, "tty"   },
 { MEM_PROC_NR,   0, SRV_F,  4,      2, 0,     SRV_T, SYS_M, MEM_C, "mem"   },
 { LOG_PROC_NR,   0, SRV_F,  4,      2, 0,     SRV_T, SYS_M, DRV_C, "log"   },
 { INIT_PROC_NR,  0, USR_F,  8, USER_Q, 0,     USR_T, USR_M,     0, "init"  },
};
```

- Το κβάντο (quantum) πρέπει να είναι θετικό – ορίζεται σε tick του ρολογιού
- Οι εγγραφές πρέπει να εμφανίζονται με την ίδια σειρά με το image -- πρώτα πρέπει να είναι τα kernel tasks

Γενικά θέματα

- Ο χρονοπρογραμματισμός γίνεται για κάθε ουρά εκ περιτροπής
- Όταν η τρέχουσα διεργασία χρησιμοποιήσει όλο το quantum που της αναλογεί, μεταφέρεται στο τέλος της ουράς και ανανεώνεται το quantum
- Αν όμως η διεργασία μπλοκάρει, όταν ενεργοποιηθεί ξανά, τοποθετείτε στην αρχή της ουράς
  - Χρησιμοποιεί το υπολειπόμενο quantum
- Χρησιμοποιούμε τον πίνακα *rdy_tail* για να προσθέτουμε γρήγορα/εύκολα μια διεργασία στο τέλος της ουράς
- Όταν μια διεργασία μπλοκάρει ή τερματίσει τότε η διεργασία αφαιρείτε από την ουρά
- Οι ουρές περιέχουν μόνο τις διεργασίες που 'τρέχουν'

Αλγόριθμος Χρονοπρογραμματισμού

- Ο αλγόριθμος είναι σχετικά απλός:
  1. Εντόπισε την ουρά με την μεγαλύτερη προτεραιότητα που δεν είναι άδεια
  2. Επέλεξε την διεργασία που είναι στην αρχή της ουράς
  3. Ενεργοποίησε την διεργασία
- Αν δεν επιλεχθεί κάποια διεργασία, τότε εκτελείτε η διεργασία IDLE
- Ο εντοπισμός της ουράς / διεργασίας υλοποιείται από την συνάρτηση *pick_proc*
- Οι συναρτήσεις *enqueue* και *dequeue* χρησιμοποιούνται για την επεξεργασία των ουρών
  - επί της ουσίας, υλοποιούν 'βασικές' λειτουργίες σε ουρές
  - χρησιμοποιούν την συνάρτηση *sched* που αποφασίζει σε ποία ουρά πρέπει να προστεθεί η διεργασία
  - Ορίζονται στο αρχείο */usr/src/kernel/proc.c*

Συνάρτηση pick_proc

```
PRIVATE void pick_proc() {
  register struct proc *rp;
  int q;

  /* Check each of the scheduling queues for ready
  for (q=0; q < NR_SCHED_QUEUES; q++) {
      if ( (rp = rdy_head[q]) != NIL_PROC) {
          next_ptr = rp;      /* run process 'rp'
          if (priv(rp)->s_flags & BILLABLE)
              bill_ptr = rp;    /* bill for system t
          return;
      }
  }
}
```

Συνάρτηση sched

```
PRIVATE void sched(rp, queue, front)
register struct proc *rp; /* process to be schedule
int *queue;               /* return: queue to use *
int *front;               /* return: front or back
{
  int time_left = (rp->p_ticks_left > 0);
  if (! time_left) {
      rp->p_ticks_left = rp->p_quantum_size;
      if (rp->p_priority < (IDLE_Q-1))
          rp->p_priority += 1;  /* lower priority *
  }
  // if there is time left, add to the front
  *queue = rp->p_priority;
  *front = time_left;
}
```

Συνάρτηση enqueue (1)

```
/* this process is now runnable */
PRIVATE void enqueue(rp)
register struct proc *rp;
{
  int q;      /* scheduling queue to use */
  int front; /* add to front or back */

#if DEBUG_SCHED_CHECK
  check_runqueues("enqueue");
  if (rp->p_ready) kprintf("enqueue() already ready
#endif

  /* Determine where to insert to process. */
  sched(rp, &q, &front);
```

Συνάρτηση enqueue (2)

```
/* Now add the process to the queue. */
 if (rdy_head[q] == NIL_PROC) {       /* add to emp
     rdy_head[q] = rdy_tail[q] = rp; /* create a n
     rp->p_nextready = NIL_PROC;      /* mark new e
 }
 else if (front) { /* add to head of queue */
     rp->p_nextready = rdy_head[q];  /* chain head
     rdy_head[q] = rp;                /* set new qu
 }
 else {  /* add to tail of queue */
     rdy_tail[q]->p_nextready = rp;  /* chain tail
     rdy_tail[q] = rp;                /* set new qu
     rp->p_nextready = NIL_PROC;      /* mark new e
 }
```

Συνάρτηση enqueue (3)

```
  /* Now select the next process to run. */
  pick_proc();

#if DEBUG_SCHED_CHECK
  rp->p_ready = 1;
  check_runqueues("enqueue");
#endif
}
```

Συνάρτηση dequeue (1)

```
/* this process is no longer runnable */
PRIVATE void dequeue(rp)
register struct proc *rp;
{
/* A process must be removed from the scheduling
 * queues, for example, because it has blocked.
 * If the currently active process is removed,
 * a new process is picked to run by calling
 * pick_proc().
 */
  register int q = rp->p_priority; /* queue to use
  register struct proc **xpp;      /* iterate over
  register struct proc *prev_xp;
```

Συνάρτηση dequeue (2)

```
/* Side-effect for kernel:
 * check if the task's stack still is ok? */
if (iskernelp(rp)) {
  if (*priv(rp)->s_stack_guard != STACK_GUARD)
    panic("stack overrun by task", proc_nr(rp));
}

#if DEBUG_SCHED_CHECK
  check_runqueues("dequeue");
  if (! rp->p_ready) kprintf("dequeue() already unr
#endif
```

Συνάρτηση dequeue (3)

```
// Now make sure that the process is not in its r
prev_xp = NIL_PROC;
for (xpp = &rdy_head[q];
     *xpp != NIL_PROC;
     xpp = &(*xpp)->p_nextready) {
    if (*xpp == rp) { /* found process to remove
        *xpp = (*xpp)->p_nextready; /* replace wi
        if (rp == rdy_tail[q]) /* queue tail remo
            rdy_tail[q] = prev_xp;  /* set new ta
        if (rp == proc_ptr || rp == next_ptr) /*
            pick_proc(); /* pick new process to r
        break;
    }
    prev_xp = *xpp; /* save previous in chain */
}
```

Συνάρτηση dequeue (4)

```
#if DEBUG_SCHED_CHECK
  rp->p_ready = 0;
  check_runqueues("dequeue");
#endif
}
```

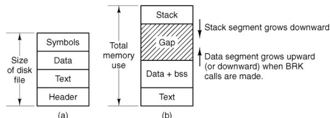
- Περιοδικά ελέγχουμε την ακεραιότητα της μνήμης του πυρήνα
- Όταν αφαιρούμε μια διεργασία που εκτελείτε στην περιοχή του πυρήνα (kernel space)
- Εάν βρεθεί πρόβλημα, είναι αδύνατο να το διορθώσουμε ...

Διαχείριση Μνήμης – Θέματα Σχεδιασμού / Αρχιτεκτονικής

- Η διαχείριση της μνήμης στο Minix 3 είναι ιδιαίτερα απλή
  - Δεν υλοποιεί μηχανισμό σελιδοποίησης (paging)
  - Υλοποιεί έναν απλό μηχανισμό swapping -- απενεργοποιημένο
- Συγκεκριμένοι λόγοι για αυτή την στρατηγική
  1. Απλούστευση συστήματος – μείωση κώδικα
  2. Απόφαση προηγούμενων εκδόσεων
  3. Ευκολία στην μεταφορά σε ενσωματωμένα συστήματα
- Αν θελήσουμε να υλοποιήσουμε έναν μηχανισμό σελιδοποίησης ή κάποιον σύνθετο μηχανισμό swapping
  - Ο διαχωρισμός των λειτουργιών στον διαχειριστή διεργασιών και στον πυρήνα διευκολύνει σε μεγάλο βαθμό
  - Ο πυρήνας ασχολείται με θέματα ανάθεσης μνήμης, δημιουργίας διεργασιών κλπ. σε χαμηλό επίπεδο
  - Ο διαχειριστής διεργασιών ασχολείται με την υλοποίηση των μηχανισμών – δεν χρειάζεται να κάνουμε αλλαγές στον πυρήνα

Τμήματα Διεργασιών

- Οι διεργασίες στο Minix 3 χωρίζονται σε τρία τμήματα
  1. text -- ο κώδικα (‘εκτελέσιμο’), δεν μεταβάλλεται
  2. data -- τα δεδομένα
  3. stack -- το stack εντολών
- Το τμήμα stack μεταβάλλεται προς το τμήμα data και αντίστροφα
- Το αρχικό μέγεθος ορίζεται από το αρχείο του εκτελέσιμου (στην επικεφαλίδα)
  - Με την εντολή *chmem* μπορούμε να μεταβάλουμε το αρχικό μέγεθος



Τμήματα Διεργασιών – Θέματα Υλοποίησης (1)

- Το μέγεθος του κάθε τμήματος μετριέται σε clicks
  - Κάθε click είναι 1024 bytes
- Στο αρχείο */usr/src/include/minix/type.h* ορίζεται η δομή *mem_map* που περιγράφει το κάθε τμήμα
  - Εικονική θέση του τμήματος στην μνήμη (σε clicks)
  - Πραγματική θέση του τμήματος στην μνήμη (σε clicks)
  - Μέγεθος τμήματος(σε clicks)

```
struct mem_map {
  vir_clicks mem_vir;     /* virtual address */
  phys_clicks mem_phys;   /* physical address */
  vir_clicks mem_len;     /* length */
};
```

- Η εικονική θέση και το μέγεθος μετριούνται σε *unsigned int*, η πραγματική θέση σε *unsigned long*

Τμήματα Διεργασιών – Θέματα Υλοποίησης (2)

- Η πληροφορία για τη θέση του κάθε τμήματος και το μέγεθος διατηρείται στον πυρήνα και στον διαχειριστή διεργασιών
- Ορίζεται ως ένας πίνακας
  - Η θέση 0 περιέχει το text (T)
  - Η θέση 1 περιέχει το data (D)
  - Η θέση 2 περιέχει το stack (S)

Πληροφορία στον Πυρήνα

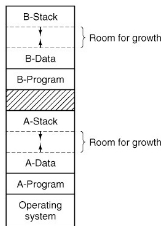
```
struct mem_map p_memmap[NR_LOCAL_SEGS];
```

Πληροφορία στον Διαχειριστή Διεργασιών

```
struct mem_map mp_seg[NR_LOCAL_SEGS];
```

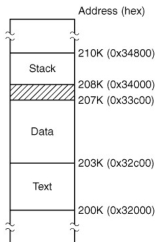
- Η σταθερά NR_LOCAL_SEGS είναι 3 και ορίζεται στο αρχείο */usr/src/include/minix/const.h*
  - Επίσης ορίζει τις 3 βοηθητικές σταθερές (T, D, S)

Τμήματα Διεργασιών – Μέγεθος Τμημάτων



- Κατά την αρχικοποίηση της διεργασίας ορίζονται τα 3 τμήματα (μέγεθος, θέση)
- Μπορεί ένα τμήμα να χρειαστεί περισσότερο χώρο (αφορά το data, stack)
  - Λογικό για το μεγαλύτερο μέρος των προγραμμάτων
- Αφήνουμε ένα ‘κενό’ μεταξύ data -- stack
- Μειώνουμε το κόστος ‘μεταφοράς’ των τμημάτων σε νέες θέσεις μνήμης με μεγαλύτερο ελεύθερο χώρο

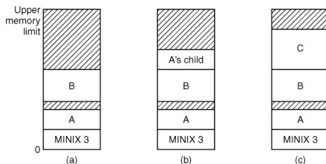
Τμήματα Διεργασιών – Παράδειγμα



- Τα τρία τμήματα της διεργασίας:
  1. text -- μέγεθος 3K
  2. data -- μέγεθος 4K
  3. stack -- μέγεθος 2K
- Κενό μεταξύ data -- stack 1K
- Συνολικό μέγεθος 10K
- Περιεχόμενα πίνακα:

	Εικ.	Πραγ.	Μεγ.
Text	0	0xc8	0x3
Data	0	0xcb	0x4
Stack	0x5	0xd0	0x2

Εκτέλεση Εντολών – Δημιουργία Νέων Διεργασιών



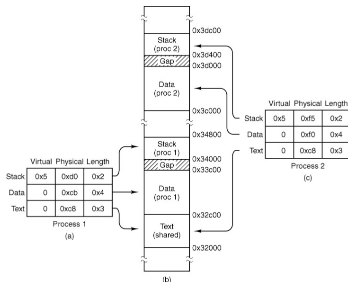
- Θέλουμε να εκτελέσουμε μια νέα εντολή
- Αρχικά χρησιμοποιούμε την κλήση του συστήματος fork
  - Αντιγράφονται τα τμήματα μνήμης της A
- Αμέσως μετά χρησιμοποιούμε την κλήση του συστήματος exec
  - Αντικαθιστώνται τα τμήματα μνήμης από αυτά της C

Κοινά Τμήματα Μνήμης

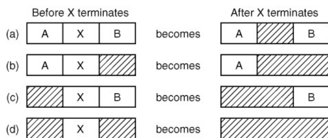
- Στα αλληλεπιδραστικά συστήματα μια διεργασία μπορεί να 'τρέχει' πολλές φορές
  - ... αντίγραφα του ίδιου κώδικα στην μνήμη
  - π.χ. χρησιμοποιώντας την κλήση του συστήματος fork
- Στην ουσία το τμήμα text είναι κοινό για όλες τις διεργασίες
- Θέλουμε να εντοπίσουμε κατά πόσο μια διεργασία χρησιμοποιεί το ίδιο τμήμα text
  - Ανατρέχουμε στο εκτελέσιμο της διεργασίας
  - Αν δεν έχουν γίνει αλλαγές, πρόκειται για αντίγραφο

```
/* File identification for sharing. */
ino_t mp_ino;    // inode number of file
dev_t mp_dev;    // device number of file system
time_t mp_ctime; // inode changed time
```

Κοινά Τμήματα Μνήμης – Παράδειγμα

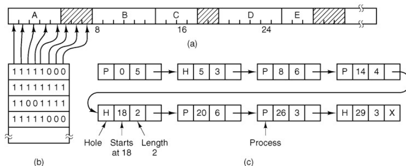


Ελεύθερη Μνήμη – Οπές Μνήμης (Memory Holes)



- Όταν τερματίζει μια διεργασία ελευθερώνεται η μνήμη
  - Χρήση της κλήσης του συστήματος exit
- Αντίστοιχα όταν δημιουργείται μια νέα διεργασία πρέπει να εντοπίσουμε ένα αρκετά μεγάλο ελεύθερο τμήμα στην μνήμη
  - Καταγράφουμε τις ελεύθερες θέσεις μνήμης – οπές (holes)

Διαχείριση Οπών Μνήμης



- Στην θεωρία (Λ.Σ. Ι) μελετήσαμε 2 μηχανισμούς διατήρησης των οπών μνήμης – έστω η παραπάνω κατάσταση της μνήμη (a)
  - Χρήση bitmap (b) ή χρήση Linked List (c)
- Το Minix 3 χρησιμοποιεί μια λίστα οπών (Hole List)

Λίστα Οπών Μνήμης – Εσωτερικές Δομές

- Στο αρχείο */usr/src/include/minix/type.h* ορίζεται η δομή *hole* που περιγράφει το κάθε οπή

```
struct hole {
  struct hole *h_next; // pointer to next entry
  phys_clicks h_base;  // where does the hole be
  phys_clicks h_len;   // how big is the hole?
};
```

- Στο αρχείο */usr/src/servers/pm/alloc.c* γίνεται η διαχείριση των οπών και υλοποιούνται οι μηχανισμοί διαχείρισης της μνήμης
- Εσωτερικά διατηρούνται ορισμένες μεταβλητές

```
#define NIL_HOLE (struct hole *) 0
PRIVATE struct hole *hole_head; // head of list
PRIVATE struct hole hole[_NR_HOLES];
```

Μηχανισμοί Διαχείριση Μνήμης

Το αρχείο */usr/src/servers/pm/alloc.c* προσφέρει τις εξής συναρτήσεις:

1. *alloc_mem* -- Δέσμευση μνήμης συγκεκριμένου μεγέθους
2. *free_mem* -- Αποδέσμευση μνήμης
3. *init_mem* -- Αρχικοποίηση μνήμης κατά την εκκίνηση του διαχειριστή διεργασιών

```
PUBLIC phys_clicks alloc_mem(phys_clicks clicks);

PUBLIC void free_mem(base, clicks);
phys_clicks base;   // base address of block to fre
phys_clicks clicks; // number of clicks to free

PUBLIC void mem_init(chunks, free);
struct memory *chunks; // list of free memory chunk
phys_clicks *free;     // memory size summaries
```

Ανάθεση Μνήμης

- Η ανάθεση μνήμης είναι απλή – χρησιμοποιείται αποκλειστικά τις κλήσεις του συστήματος *fork* και *exec*
  - Η μνήμη που έχει ανατεθεί δεν μπορεί να μεγαλώσει/μικρύνει κατά την διάρκεια ζωής της διεργασίας
- Εάν πρόκειται για μια διεργασία που `μοιράζεται' το τμήμα text με κάποια άλλη, τότε η ανάθεση μνήμης γίνεται μόνο για τα τμήματα data και stack
- Η συνάρτηση *alloc_mem* ανατρέχει την λίστα οπών έως ότου βρει την πρώτη εγγραφή που είναι αρκετά μεγάλη για να `χωρέσει' το μέγεθος της μνήμης που θέλουμε να αναθέσουμε
- Αν δεν βρεθεί καμία εγγραφή (αρκετά μεγάλη) τότε
  - Αν είναι ενεργοποιημένος ο μηχανισμός swapping -- προσπαθεί να κάνει swap out κάποια διεργασία και επαναλαμβάνει την αναζήτηση έως ότου βρεθεί μια αρκετά μεγάλη οπή
  - Αν δεν μπορεί να βρεθεί (συνεχόμενος) ελεύθερος χώρος τότε επιστρέφει *NO_MEM*

Ανάθεση Μνήμης – Συνάρτηση alloc_mem (1)

```
PUBLIC phys_clicks alloc_mem(clicks)
phys_clicks clicks; // amount of memory requested
{
  register struct hole *hp, *prev_ptr;
  phys_clicks old_base;
  do {
    // try to allocate a block of memory
    // by searching the list of holes
    // if found, return a pointer to the block
  } while (swap_out());
    /* try to swap some other process out
     * (if swaping is enabled) */
  return(NO_MEM);
}
```

Ανάθεση Μνήμης – Συνάρτηση alloc_mem (2)

```
prev_ptr = NIL_HOLE;
hp = hole_head;
while (hp != NIL_HOLE && hp->h_base<swap_base){
  if (hp->h_len >= clicks) {
    /* We found a hole that is big enough.
     * Use it.
     * Return the start address of the
     * acquired block. */
    return(old_base);
  }

  // Nothing found, continue to next hole
  prev_ptr = hp;
  hp = hp->h_next;
}
```

Ανάθεση Μνήμης – Συνάρτηση alloc_mem (3)

```
// We found a hole that is big enough. Use it.
old_base = hp->h_base; // remember where it started
hp->h_base += clicks;  // bite a piece off
hp->h_len -= clicks;   // ditto

// Remember new high watermark of used memory
if(hp->h_base > high_watermark)
  high_watermark = hp->h_base;

// Delete the hole if used up completely
if (hp->h_len == 0) del_slot(prev_ptr, hp);

// Return the start address of the acquired block
return(old_base);
```

Απελευθέρωση Μνήμης – Βοηθητική Συνάρτηση del_slot

```
PRIVATE void del_slot(prev_ptr, hp)
register struct hole *prev_ptr; // hole entry ahead
register struct hole *hp; // hole entry to be remov
{
/* Remove an entry from the hole list.
 * request to allocate memory removes a hole
 * in its entirety */
  if (hp == hole_head) hole_head = hp->h_next;
  else prev_ptr->h_next = hp->h_next;

  hp->h_next = free_slots;
  hp->h_base = hp->h_len = 0;
  free_slots = hp;
}
```

Απελευθέρωση Μνήμης

- Η απελευθέρωση μνήμης είναι επίσης απλή
  - Χρησιμοποιείται από την κλήση του συστήματος *exit*
  - 'Οταν είναι ενεργοποιημένος ο μηχανισμός swapping
- Προσπαθεί να εντοπίσει κάποια υπάρχουσα οπή που είναι 'δίπλα' στην θέση μνήμης που πρόκειται να απελευθερωθεί
  - Συγχώνευση της θέσης μνήμης με την υπάρχουσα οπή
- Αλλιώς εισάγει μια νέα εγγραφή στην λίστα οπών με την θέση μνήμης που απελευθερώθηκε
- Ο πίνακας των οπών έχει συγκεκριμένο μέγεθος
  - Το μέγεθος ορίζεται από την σταθερά _NR_HOLES ορίζεται σε (2*_NR_PROCS+4) – αρχείο */usr/src/include/minix/sys_config.h*
  - Αν και είναι ικανοποιητικά μεγάλος (για τις απαιτήσεις του Minix 3) – μπορεί να γεμίσει
  - Σε αυτή την περίπτωση προκύπτει ένα kernel panic

Απελευθέρωση Μνήμης – Συνάρτηση free_mem (1)

```
PUBLIC void free_mem(base, clicks)
phys_clicks base;   // base address of block to fre
phys_clicks clicks; // number of clicks to free
{
  register struct hole *hp, *new_ptr, *prev_ptr;

  if (clicks == 0) return;

  if ( (new_ptr = free_slots) == NIL_HOLE)
   panic(__FILE__,"hole table full", NO_NUM);

  new_ptr->h_base = base;
  new_ptr->h_len = clicks;
  free_slots = new_ptr->h_next;
  hp = hole_head;
```

Απελευθέρωση Μνήμης – Συνάρτηση free_mem (2)

```
/* If this block's address is numerically
 * less than the lowest hole currently
 * available, or if no holes are currently
 * available, put this hole on the front
 * of the hole list.
 */
if (hp == NIL_HOLE || base <= hp->h_base) {
  // Block to be freed goes on front of the hole
  new_ptr->h_next = hp;
  hole_head = new_ptr;
  merge(new_ptr);
  return;
}
```

Απελευθέρωση Μνήμης – Συνάρτηση free_mem (3)

```
    // Block to be returned does not go on
    // front of hole list
    prev_ptr = NIL_HOLE;
    while (hp != NIL_HOLE && base > hp->h_base) {
      prev_ptr = hp;
      hp = hp->h_next;
    }

    // We found where it goes.
    // Insert block after 'prev_ptr'
    new_ptr->h_next = prev_ptr->h_next;
    prev_ptr->h_next = new_ptr;
    merge(prev_ptr);
  }
```

Απελευθέρωση Μνήμης – Βοηθητική Συνάρτηση merge (1)

```
PRIVATE void merge(hp)
register struct hole *hp; /* hole to merge with
{
  // Check for contiguous holes and merge any found
  register struct hole *next_ptr;
  // 'hp' points to the last hole, merging impossib
  if ( (next_ptr = hp->h_next) == NIL_HOLE) return;

  if (hp->h_base+hp->h_len==next_ptr->h_base) {
    /* first one gets second one's mem */
    hp->h_len += next_ptr->h_len;
    del_slot(hp, next_ptr);
  } else {
    hp = next_ptr;
  }
```

Απελευθέρωση Μνήμης – Βοηθητική Συνάρτηση merge (2)

```
    /* If 'hp' now points to the last hole,
     * return; otherwise, try to absorb its
     * successor into it.
     */
    if ( (next_ptr = hp->h_next) == NIL_HOLE)
      return;

    if (hp->h_base+hp->h_len==next_ptr->h_base) {
      hp->h_len += next_ptr->h_len;
      del_slot(hp, next_ptr);
    }
  }
```

Γενικά Θέματα Σημάτων

- Ενας βασικός τρόπος επικοινωνίας των διεργασιών είναι η χρήση σημάτων
  - Μεταφέρουν πληροφορία σε διεργασίες που δεν περιμένουν κάποια είσοδο
  - Χαρακτηρίζονται ως τα interrupt λογισμικού
- Μελετήσαμε τα θέματα προγραμματισμού στο προηγούμενο εξάμηνο
  - 2η άσκηση Λ.Σ.Ι
- Το MINIX 3 ακολουθεί το πρότυπο POSIX
  - Ο κώδικας που αναπτύχθηκε στα πλαίσια της 2ης άσκησης των Λ.Σ.Ι μπορεί να μεταφερθεί στο MINIX 3 με `σχετική' ευκολία
  - Όπως και με τα υπόλοιπα ζητήματα, η υλοποίηση των σημάτων είναι μινιμαλιστική

Ορισμός Σημάτων -- signal.h (1)

```
/* hangup */
#define SIGHUP              1
/* interrupt (DEL) */
#define SIGINT              2
/* quit (ASCII FS) */
#define SIGQUIT             3
/* illegal instruction */
#define SIGILL              4
/* trace trap (not reset when caught) */
#define SIGTRAP             5
/* IOT instruction */
#define SIGABRT             6
/* bus error */
#define SIGBUS              7
```

Ορισμός Σημάτων -- signal.h (2)

```
/* floating point exception */
#define SIGFPE              8
/* kill (cannot be caught or ignored) */
#define SIGKILL             9
/* user defined signal # 1 */
#define SIGUSR1            10
/* segmentation violation */
#define SIGSEGV            11
/* user defined signal # 2 */
#define SIGUSR2            12
/* write on a pipe with no one to read it */
#define SIGPIPE            13
/* alarm clock */
#define SIGALRM            14
```

Ορισμός Σημάτων -- signal.h (3)

```
/* software termination signal from kill */
#define SIGTERM            15
/* EMT instruction */
#define SIGEMT   16
/* child process terminated or stopped */
#define SIGCHLD            17
/* window size has changed */
#define SIGWINCH        21
/* continue if stopped */
#define SIGCONT            18
/* stop signal */
#define SIGSTOP            19
/* interactive stop signal */
#define SIGTSTP            20
```

Ορισμός Σημάτων -- signal.h (4)

```
/* background process wants to read */
#define SIGTTIN            22
/* background process wants to write */
#define SIGTTOU            23

/* new kernel message */
#define SIGKMESS       29
/* kernel signal pending */
#define SIGKSIG        30
/* kernel shutting down */
#define SIGKSTOP        31
```

Βασική Λειτουργία Σημάτων

- Όταν μια διεργασία λάβει ένα σήμα, το σύστημα ορίζει κάποιες προκαθορισμένες αντιδράσεις
- Μια διεργασία μπορεί να ζητήσει κάποια σήματα να αγνοηθούν
- Μια διεργασία μπορεί να αλλάξει την προκαθορισμένη αντίδραση για ένα σήμα ορίζοντας έναν signal handler
- Επομένως υπάρχουν τρεις βασικές φάσεις για την διαχείριση των σημάτων
  1. Προετοιμασία: Ρυθμίσεις σημάτων και αντιστοίχηση signal handler
  2. Απάντηση: Χρήση signal handler όταν ληφθεί ένα σήμα
  3. Επαναφορά: επιστροφή της διεργασίας στην 'κανονική' ροή εκτέλεσης
- Το Λ.Σ. αναλαμβάνει να εκτελέσει τον signal handler και να επαναφέρει την διεργασία στην προηγούμενη ροή εκτέλεσης

Προετοιμασία Σημάτων (1)

- Υπάρχουν διάφορες κλήσεις του συστήματος για την ρύθμιση της αντίδρασης μιας διεργασίας σε εισερχόμενα σήματα
  - sigaction – ορισμός signal handler, επαναφορά προκαθορισμένης αντίδρασης, 'απενεργοποίηση' σήματος
  - sigprocmask – μπλοκάρισμα σήματος
- Για κάθε διεργασία αντιστοιχούν διάφορες μεταβλητές τύπου sigset_t (στον πίνακα των διεργασιών)
- Ορίζεται στο αρχείο */usr/src/include/signal.h* ως `typedef unsigned long sigset_t;`
- Χρησιμοποιείται ως bitmap όπου κάθε bit αντιστοιχεί σε ένα σήμα
  - π.χ., η μεταβλητή *mp_catch* ορίζει κατά πόσο το συγκεκριμένο σήμα είναι 'ενεργοποιημένο' για την διεργασία
  - π.χ., η μεταβλητή *mp_ignore* ορίζει κατά πόσο το συγκεκριμένο σήμα είναι 'απενεργοποιημένο' για την διεργασία

Προετοιμασία Σημάτων (2)

- Επίσης, για κάθε διεργασία, για κάθε σήμα, αντιστοιχεί μια μεταβλητή τύπου sigaction (στον πίνακα των διεργασιών)

```
struct sigaction {
  /* SIG_DFL, SIG_IGN, or pointer to function */
  __sighandler_t sa_handler;
  /* signals to be blocked during handler */
  sigset_t sa_mask;
  /* special flags */
  int sa_flags;
};
```

- O signal handler ορίζεται ως

```
typedef void _PROTOTYPE( (*__sighandler_t),
(int) );
```

Προετοιμασία Σημάτων – Παράδειγμα

Αρχείο testsignal.c

```
void catch_int(int sig_num) {
  printf("Don't do that...");
}
int main() {
  struct sigaction sig;
  sigset_t sset;
  sigemptyset(&sset);
  sig.sa_flags = 0;
  sig.sa_handler = catch_int;
  sig.sa_mask = sset;
  sigaddset(&sig.sa_mask, SIGINT);
  sigaction(SIGINT, &sig, NULL);
}
```

Αποστολή Σημάτων (1)

- Όταν δημιουργείται ένα νέο σήμα διάφορα τμήματα του MINIX 3 ενεργοποιούνται για την διαχείριση και αποστολή του σήματος στην διεργασία
- Η διαδικασία ξεκινάει από τον διαχειριστή διεργασιών
  - Εντοπίζει την διεργασία που πρέπει να παραλάβει το σήμα
  - Ελέγχει κατά πόσο η διεργασία έχει 'ενεργοποιήσει' το σήμα
- Εφόσον το σήμα είναι ενεργοποιημένο ξεκινάει η διαδικασία παράδοσης του σήματος
  - Η κανονική ροή εκτέλεσης πρέπει να διακοπεί
  - Πληροφορίες σχετικά με την ροή εκτέλεσης αποθηκεύονται στο stack (εφόσον υπάρχει χώρος)
  - Αυτά τα βήματα γίνονται από τον διαχειριστή διεργασιών

Αποστολή Σημάτων (2)

- Η τελική παράδοση του σήματος γίνεται από το system task
  - Διακόπτεται η κανονική ροή εκτέλεσης
  - Αλλάζει ο program counter έτσι ώστε ροή εκτέλεσης να συνεχίσει με τον signal handler
  - Όταν ολοκληρωθεί η εκτέλεση του signal handler γίνεται μια κλήση συστήματος *SIGRETURN*
- Η κλήση του συστήματος *SIGRETURN*
  - Ενεργοποιεί τον διαχειριστή διεργασιών για την επαναφορά του stack
  - Ενεργοποιεί τον πυρήνα για την αλλαγή του program counter έτσι ώστε η ροή εκτέλεσης να επανέλθει στην προηγούμενη κατάσταση
- Η παραπάνω διαδικασία μπορεί να χρειαστεί να εκτελεστεί πολλαπλές φορές αν ένα σήμα αφορά μια ομάδα/ιεραρχία διεργασιών

Λίστα Σημάτων (1)

Signal	Description	Generated by
SIGHUP	Hangup	KILL system call
SIGINT	Interrupt	TTY
SIGQUIT	Quit	TTY
SIGILL	Illegal instruction	Kernel (*)
SIGTRAP	Trace trap	Kernel (M)
SIGABRT	Abnormal termination	TTY
SIGFPE	Floating point exception	Kernel (*)
SIGKILL	Kill (cannot be caught or ignored)	KILL system call

* Βασίζονται στην υποστήριξη του hardware

M Δεν ορίζονται από το POSIX αλλά υποστηρίζονται για λόγους συμβατότητας

Λίστα Σημάτων (2)

Signal	Description	Generated by
SIGUSR1	User-defined signal #1	Not supported
SIGUSR2	User defined signal #2	Not supported
SIGSEGV	Segmentation violation	Kernel (*)
SIGPIPE	Write on a pipe with no one to read it	FS
SIGALRM	Alarm clock, timeout	PM
SIGTERM	Software termination signal from kill	KILL system call
SIGCHLD	Child process terminated or stopped	PM

* Βασίζονται στην υποστήριξη του hardware

M Δεν ορίζονται από το POSIX αλλά υποστηρίζονται για λόγους συμβατότητας

Λίστα Σημάτων (3)

Signal	Description	Generated by
SIGCONT	Continue if stopped	Not supported
SIGSTOP	Stop signal	Not supported
SIGTSTP	Interactive stop signal	Not supported
SIGTTIN	Background process wants to read	Not supported
SIGTTOU	Background process wants to write	Not supported
SIGKMESS	Kernel message	Kernel
SIGKSIG	Kernel signal pending	Kernel
SIGKSTOP	Kernel shutting down	Kernel

- Τα σήματα που δημιουργεί ο πυρήνας είναι ειδικά σήματα που ενημερώνουν τις διεργασίες του συστήματος για ειδικά γεγονότα

Ορισμός Βασικών signal handler -- signal.h

```
/* Macros used as function pointers. */

/* default signal handling */
#define SIG_DFL    ((__sighandler_t)  0)
/* ignore signal */
#define SIG_IGN    ((__sighandler_t)  1)
/* block signal */
#define SIG_HOLD   ((__sighandler_t)  2)
/* catch signal */
#define SIG_CATCH  ((__sighandler_t)  3)
```

Παράμετροι Σημάτων sa_flags -- signal.h

```
/* deliver signal on alternate stack */
#define SA_ONSTACK   0x0001
/* reset signal handler when signal caught */
#define SA_RESETHAND 0x0002
/* don't block signal while catching it */
#define SA_NODEFER   0x0004
/* automatic system call restart */
#define SA_RESTART   0x0008
/* extended signal handling */
#define SA_SIGINFO   0x0010
/* don't create zombies */
#define SA_NOCLDWAIT 0x0020
/* don't receive SIGCHLD when child stops */
#define SA_NOCLDSTOP 0x0040
```

Δομή Πίνακα Διεργασιών

```
// Signal handling information
sigset_t mp_ignore;   // 1 ignore the signal
sigset_t mp_catch;    // 1 catch the signal
sigset_t mp_sig2mess; // 1 transform into notify
sigset_t mp_sigmask;  // signals to be blocked
sigset_t mp_sigmask2; // saved copy of mp_sigmask
sigset_t mp_sigpending; // pending signals
// as in sigaction(2)
struct sigaction mp_sigact[_NSIG + 1];
// address of C library __sigreturn function
vir_bytes mp_sigreturn;
// watchdog timer for alarm(2)
struct timer mp_timer;
```

Βοηθητική Συνάρτηση *sigaction*

- Ορίζεται στο αρχείο */usr/src/lib/posix/_sigaction.c*

```
PUBLIC int sigaction(sig, act, oact)
int sig;
_CONST struct sigaction *act;
struct sigaction *oact;
{
  message m;
  m.m1_i2 = sig;
  m.m1_p1 = (char *) act;
  m.m1_p2 = (char *) oact;
  m.m1_p3 = (char *) __sigreturn;
  return(_syscall(MM, SIGACTION, &m));
}
```

Κλήση Συστήματος *SIGACTION* (1)

```
PUBLIC int do_sigaction() {
  int r;
  struct sigaction svec;
  struct sigaction *svp;
  if (m_in.sig_nr == SIGKILL) return(OK);
  if (m_in.sig_nr < 1 || m_in.sig_nr > _NSIG)
    return (EINVAL);
  svp = &mp->mp_sigact[m_in.sig_nr];
  if ((struct sigaction *) m_in.sig_osa !=
      (struct sigaction *) NULL) {
    r = sys_datacopy(PM_PROC_NR,(vir_bytes) svp,
             who_e, (vir_bytes) m_in.sig_osa,
             (phys_bytes) sizeof(svec));
    if (r != OK) return(r);
  }
```

Κλήση Συστήματος *SIGACTION* (2)

```
  if ((struct sigaction *) m_in.sig_nsa ==
      (struct sigaction *) NULL)
     return(OK);

  /* Read in the sigaction structure. */
  r = sys_datacopy(who_e, (vir_bytes) m_in.sig_nsa,
               PM_PROC_NR, (vir_bytes) &svec,
               (phys_bytes) sizeof(svec));
  if (r != OK) return(r);

  if (svec.sa_handler == SIG_IGN) {
    sigaddset(&mp->mp_ignore, m_in.sig_nr);
    sigdelset(&mp->mp_sigpending, m_in.sig_nr);
    sigdelset(&mp->mp_catch, m_in.sig_nr);
    sigdelset(&mp->mp_sig2mess, m_in.sig_nr);
```

Κλήση Συστήματος *SIGACTION* (3)

```
  } else if (svec.sa_handler == SIG_DFL) {
      sigdelset(&mp->mp_ignore, m_in.sig_nr);
      sigdelset(&mp->mp_catch, m_in.sig_nr);
      sigdelset(&mp->mp_sig2mess, m_in.sig_nr);
  } else if (svec.sa_handler == SIG_MESS) {
    if (! (mp->mp_flags & PRIV_PROC)) return(EPERM)
      sigdelset(&mp->mp_ignore, m_in.sig_nr);
      sigaddset(&mp->mp_sig2mess, m_in.sig_nr);
      sigdelset(&mp->mp_catch, m_in.sig_nr);
  } else {
      sigdelset(&mp->mp_ignore, m_in.sig_nr);
      sigaddset(&mp->mp_catch, m_in.sig_nr);
      sigdelset(&mp->mp_sig2mess, m_in.sig_nr);
  }
```

Κλήση Συστήματος *SIGACTION* (4)

```
  mp->mp_sigact[m_in.sig_nr].sa_handler =
     svec.sa_handler;
  sigdelset(&svec.sa_mask, SIGKILL);
  mp->mp_sigact[m_in.sig_nr].sa_mask =
     svec.sa_mask;
  mp->mp_sigact[m_in.sig_nr].sa_flags =
     svec.sa_flags;
  mp->mp_sigreturn = (vir_bytes) m_in.sig_ret;
  return(OK);
}
```

Κλήσεις Συστήματος *SIGPENDING* και *SIGPROCMASK*

- Επιτρέπεται η επεξεργασία των ρυθμίσεων των σηματων ακόμα και 'μέσα' από τον signal handler
- Η κλήση SIGPENDING επιστρέφει τα σήματα που είναι σε αναμονή
  - Δηλαδή: αυτά που πρόκειται να παραδοθούν
- Η κλήση SIGPROCMASK επιστρέφει τα σήματα που έχουν μπλοκαριστεί
  - Δηλαδή: αυτά που δεν πρόκειται να παραδοθούν

```
PUBLIC int do_sigpending() {
  mp->mp_reply.reply_mask = (long)
        mp->mp_sigpending;
  return OK;
}
```

Επικοινωνία Διεργασιών

- Το Λ.Σ. προσφέρει πολλούς τρόπους για την επικοινωνία μεταξύ διεργασιών
  - Μηνύματα
  - Σήματα
  - Sockets / Pipes
  - Διαμοιραζόμενη μνήμη (shared memory)
- Οι μηχανισμοί αυτοί είναι υψηλού επιπέδου
  - Παρέχουν σύνθετη λειτουργικότητα
  - Προϋποθέτουν συνθετων μηχανισμών
- Πως μπορούν 2 διεργασίες να επικοινωνίσουν χωρίς την χρήση αυτών των μηχανισμών;
  - ... θέλουμε ταχύτητα
  - ... θέλουμε καλήτερο έλεγχο
  - ... δεν μπορούμε να χρησιμοποιήσουμε αυτούς τους μηχανισμούς

Επικοινωνία Διεργασιών – Απ' ευθείας Αντιγραφή Μνήμης

- Ο πιό βασικός τρόπος είναι μέσω απ' ευθείας πρόσβαση στη μνήμη
  - Οι μεταβλητές διατηρούνται στο data τμήμα μιας διεργασίας
  - Αρκεί να 'διαβάσουμε' τη θέση μνήμης που βρίσκεται μια μεταβλητη ...
- 'Ομως είναι αυτος ο τρόπος ασφαλής;
- Οι διεργασίες χειρίζονται τα τμήματα μνήμης τους μεσω των εικονικών διευθύνσεων
  - πως μπορούν να εντοπίσουν την θέση μνήμης σε μια άλλη διεργασία;

Αντιγραφή Μνήμης

- Ο πυρήνας παρέχει μηχανισμούς (κλήσεις πυρήνα) για την αντιγραφή μνήμης
- Ασφάλεια:
  - Αρχιτεκτονική Μικρο-πυρήνα – μόνο οι διαχειριστές / οδηγοί μπορούν να καλέσουν αυτούς τους μηχανισμούς
- Εικονικές Διευθύνσεις:
  - Ο πυρήνας γνωρίζει τις πραγματικές διευθύνσεις που έχουν τα τμήματα των διεργασιών ...
  - Μετατρέπει τις εικονικές διευθύνσεις σε πραγματικές

Αντιγραφή Μνήμης – Κλήσεις Πυρήνα – SYS_VIRCOPY

```
_PROTOTYPE(int sys_vircopy, (int src_proc,
   int src_seg, vir_bytes src_vir,
   int dst_proc, int dst_seg, vir_bytes dst_vir,
   phys_bytes bytes));
```

- Αντιγραφή από – διεργασία `πηγή' (source)
  - src_proc -- θέση διεργασίας στον πίνακα διεργασιών
  - src_seg -- τμήμα μνήμης διεργασία (text, data, stack)
  - src_vir -- εικονική θέση μνήμης στη διεργασία
- Αντιγραφή πρός – διεργασία `προορισμός' (destination)
  - dst_proc -- θέση διεργασίας στον πίνακα διεργασιών
  - dst_seg -- τμήμα μνήμης διεργασία (text, data, stack)
  - dst_vir -- εικονική θέση μνήμης στη διεργασία
- bytes -- Πλήθος bytes προς αντιγραφή

Βοηθητική Συνάρτηση -- sys_vircopy (1)

/usr/src/lib/syslib/sys_vircopy.c

```
PUBLIC int sys_vircopy(src_proc, src_seg, src_vir,
                       dst_proc, dst_seg, dst_vir,
                       bytes)
int src_proc;       /* source process */
int src_seg;        /* source memory segment */
vir_bytes src_vir; /* source virtual address */
int dst_proc;       /* destination process */
int dst_seg;        /* destination memory segment */
vir_bytes dst_vir; /* destination virtual address */
phys_bytes bytes;   /* how many bytes */
{
...
}
```

Βοηθητική Συνάρτηση -- sys_vircopy (2)

/usr/src/lib/syslib/sys_vircopy.c

```
{
  message copy_mess;

  if (bytes == 0L) return(OK);
  copy_mess.CP_SRC_ENDPT = src_proc;
  copy_mess.CP_SRC_SPACE = src_seg;
  copy_mess.CP_SRC_ADDR = (long) src_vir;
  copy_mess.CP_DST_ENDPT = dst_proc;
  copy_mess.CP_DST_SPACE = dst_seg;
  copy_mess.CP_DST_ADDR = (long) dst_vir;
  copy_mess.CP_NR_BYTES = (long) bytes;
  return(_taskcall(SYSTASK, SYS_VIRCOPY, &copy_mess));
}
```

Αντιγραφή Μνήμης – Κλήσεις Πυρήνα (1)

- Το αρχείο `/usr/include/minix/syslib.h`
  - ορίζει `παραλλαγές' της sys_vircopy
- Λιγότερες παράμετροι ανάλογα την περίσταση
  1. sys_biosin
  2. sys_biosout
  3. sys_datacopy
  4. sys_textcopy
  5. sys_stackcopy
  6. sys_abscopy
- Όλες καταλήγουν στην do_copy `/usr/src/kernel/system/do_copy.c`

Αντιγραφή Μνήμης – Κλήσεις Πυρήνα (2)

- Στο αρχείο `/usr/src/kernel/system.h` ορίζονται οι handlers για τις κλήσεις πυρήνα

```
_PROTOTYPE( int do_copy, (message *m_ptr) );
#define do_vircopy   do_copy
#define do_physcopy  do_copy
```

- Στο αρχείο `/usr/src/kernel/system.c` γίνεται η αντιστοίχηση των handlers με τους αριθμούς των κλήσεων πυρήνα

```
map(SYS_VIRCOPY, do_vircopy);
map(SYS_PHYSCOPY, do_physcopy);
```

/usr/src/kernel/system/do_copy.c

- Θεωρεί οτι το μήνυμα είναι τύπου m5
  - τύπος (m_type) -- SYS_VIRCOPY ή SYS_PHYSCOPY
  - m5_c1 (CP_SRC_SPACE) -- source virtual segment
  - m5_l1 (CP_SRC_ADDR) -- source offset within segment
  - m5_i1 (CP_SRC_PROC_NR) -- source process number
  - m5_c2 (CP_DST_SPACE) -- destination virtual segment
  - m5_l2 (CP_DST_ADDR) -- destination offset within segment
  - m5_i2 (CP_DST_PROC_NR) -- destination process number
  - m5_l3 (CP_NR_BYTES) -- number of bytes to copy

Συνάρτηση -- do_copy (1)

- Μετατρέπει τις παραμέτρους του μηνύματος σε 2 μεταβλητές τύπου vir_addr

```
PUBLIC int do_copy(m_ptr)
register message *m_ptr;
{
  struct vir_addr vir_addr[2];
  phys_bytes bytes;

  vir_addr[_SRC_].proc_nr_e = m_ptr->CP_SRC_ENDPT;
  vir_addr[_SRC_].segment = m_ptr->CP_SRC_SPACE;
  vir_addr[_SRC_].offset = (vir_bytes) m_ptr->CP_SRC_ADD
  vir_addr[_DST_].proc_nr_e = m_ptr->CP_DST_ENDPT;
  vir_addr[_DST_].segment = m_ptr->CP_DST_SPACE;
  vir_addr[_DST_].offset = (vir_bytes) m_ptr->CP_DST_ADD
  bytes = (phys_bytes) m_ptr->CP_NR_BYTES;
```

Συνάρτηση -- do_copy (2)

- Ελέγχει τις διευθύνσεις – αν αναφέρονται σε 'πραγματικές' διεργασίες
  - Χρησιμοποιεί την συνάρτηση `isokendpt_f`
  - ... ορίζεται στο αρχείο `/usr/src/kernel/proc.c`
  - Στην ουσία μετατρέπει το αριθμό end-point σε αριθμό διεργασίας (θέση στον πίνακα διεργασιών)
  - Αν η διεργασία δεν εντοπιστεί – ή δεν είναι ζωντανή – επιστρέφει έναν αρνητικό αριθμό
- Χρησιμοποιεί την συνάρτηση `virtual_copy`
  - ορίζεται στο αρχείο `/usr/src/kernel/system.c`
  - Μετατρέπει τις εικονικές διευθύνσεις σε πραγματικές με την χρήση της συνάρτησης `umap_local`
  - Χρησιμοποιεί την συνάρτηση `phys_copy` για την τελική αντιγραφή – επίπεδο assembly (klib386.s)

Κλήση Συστήματος *KILL*

- Βασικός τρόπος για την αποστολή σημάτων

```
PUBLIC int do_kill() {
  /* Perform the kill(pid, signo) system call*/
  return check_sig(m_in.pid, m_in.sig_nr);
}
```

- Η συνάρτηση *check_sig* ανατρέχει τον πίνακα των διεργασιών για να εντοπίσει όλες τις διεργασίες που μπορούν/πρέπει να παραλάβουν το σήμα
  - Ελέγχει αν έχει δικαίωμα να λάβει το σήμα
  - Ελέγχει αν είναι διαχειριστής και δεν πρέπει να λάβει το σήμα
  - Ελέγχει αν ανήκει στην ίδια ομάδα/ιεραρχία διεργασιών
- Για κάθε διεργασία καλεί την συνάρτηση *sig_proc* για την παράδοση

Συνάρτηση *sig_proc* (1)

```
PUBLIC void sig_proc(rmp, signo)
/* pointer to the process to be signaled */
register struct mproc *rmp;
int signo; /* signal to send (1 to _NSIG) */
{
  vir_bytes new_sp;
  int s, slot, sigflags;
  struct sigmsg sm;
  slot = (int) (rmp - mproc);
  if ((rmp->mp_flags & (IN_USE | ZOMBIE)) != IN_USE
    printf("PM: signal %d sent to %s process %d\n",
        signo,
        (rmp->mp_flags & ZOMBIE) ? "zombie" : "dead"
    panic(__FILE__,"", NO_NUM);
  }
```

Συνάρτηση *sig_proc* (2)

```
if ((rmp->mp_flags & TRACED) && signo!=SIGKILL) {
  /* A traced process has special handling. */
  unpause(slot);
  stop_proc(rmp, signo); /* stop the proc */
  return;
}
/* Some signals are ignored by default. */
if (sigismember(&rmp->mp_ignore, signo)) {
  return;
}
if (sigismember(&rmp->mp_sigmask, signo)) {
  /* Signal should be blocked. */
  sigaddset(&rmp->mp_sigpending, signo);
  return;
}
```

Συνάρτηση *sig_proc* (3)

```
sigflags = rmp->mp_sigact[signo].sa_flags;
if (sigismember(&rmp->mp_catch, signo)) {
  if (rmp->mp_flags & SIGSUSPENDED)
    sm.sm_mask = rmp->mp_sigmask2;
  else
    sm.sm_mask = rmp->mp_sigmask;
  sm.sm_signo = signo;
  sm.sm_sighandler = (vir_bytes) rmp->mp_sigact[s
  sm.sm_sigreturn = rmp->mp_sigreturn;
  if ((s=get_stack_ptr(rmp->mp_endpoint,&new_sp))
    panic(__FILE__,"couldn't get new stack pointe
  sm.sm_stkptr = new_sp;
```

Συνάρτηση *sig_proc* (4)

```
/* Make room for the sigcontext and sigframe st
new_sp -= sizeof(struct sigcontext)
       + 3 * sizeof(char *) + 2 * sizeof(int);

if (adjust(rmp,rmp->mp_seg[D].mem_len,new_sp)!=
  goto doterminate;

if (OK ==(s=sys_sigsend(rmp->mp_endpoint,&sm)))
  sigdelset(&rmp->mp_sigpending, signo);
  /* If process is hanging on PAUSE, WAIT,
   * SIGSUSPEND, tty, pipe, etc., release it.
   */
  unpause(slot);
  return;
}
```

Κλήση Πυρήνα *SIGSEND*

- Τελικό στάδιο για την αποστολή ενός σήματος
- Εξυπηρετούνται μόνο οι κλήσεις που προέρχονται από τον διαχειριστή διεργασιών
- Οι πληροφορίες που αφορούν ένα μήνυμα μεταφέρονται μέσω μιας ειδικής δομής *sigmsg*
  - Αντιγράφεται η δομή στο kernel space
- Αποθηκεύει τα περιεχόμενα του *sigmsg* σε βοηθητικές δομές
  - Είναι απαραίτητα όταν θα γίνει η κλήση *SIGRETURN*
- Διακόπτει την ροή εκτέλεσης και σώζει τον program counter στο stuck
- Ενημερώνει στους registers και τον program counter σύμφωνα με τον signal handler

Βοηθητική Συνάρτηση *sys_sigsend*

```
PUBLIC int sys_sigsend(proc_nr, sig_ctxt)
int proc_nr;   /* for which process */
struct sigmsg *sig_ctxt; /* POSIX style handling */
{
  message m;
  int result;

  m.SIG_ENDPT = proc_nr;
  m.SIG_CTXT_PTR = (char *) sig_ctxt;
  result = _taskcall(SYSTASK, SYS_SIGSEND, &m);
  return(result);
}
```

Κλήση Συστήματος *SIGRETURN*

- 'Οταν ολοκληρωθεί η εκτέλεση του signal handler η διεργασία στέλνει ένα 'κρυφό' σήμα SIGRETURN
- Χρησιμοποιούμε αυτό το 'ενδιάμεσο' σήμα για την επαναφορά της ροής εκτέλεσης της διεργασίας
- Ο χειρισμός του SIGRETURN γίνεται από τον διαχειριστή διεργασιών
- Βασίζεται στην κλήση του πυρήνα *SYS_SIGRETURN*
- Επαναφέρει την ροή εκτέλεσης της διεργασίας
- 'Οταν ολοκληρωθεί η κλήση του πυρήνα *SYS_SIGRETURN* χρησιμοποιεί την συνάρτηση *check_pending*
  - Ελέγχει αν υπάρχουν σήματα σε εκκρεμότητα και τα προωθεί στις διεργασίες
  - Για κάθε συγκεκριμένη σήμα υπάρχει απλά 1 bit -- είτε έρθει 1 σήμα είτε 10 σήματα, μόνο 1 θα προωθηθεί

Κλήσεις Συστήματος *SIGPENDING* και *SIGPROCMASK*

- Επιτρέπεται η επεξεργασία των ρυθμίσεων των σηματων ακόμα και 'μέσα' από τον signal handler
- Η κλήση SIGPENDING επιστρέφει τα σήματα που είναι σε αναμονή
  - Δηλαδή: αυτά που πρόκειται να παραδοθούν
- Η κλήση SIGPROCMASK επιστρέφει τα σήματα που έχουν μπλοκαριστεί
  - Δηλαδή: αυτά που δεν πρόκειται να παραδοθούν

```
PUBLIC int do_sigpending() {
  mp->mp_reply.reply_mask = (long)
        mp->mp_sigpending;
  return OK;
}
```