

## Εργαστήριο Λειτουργικών Συστημάτων

Μάθημα 6<sup>ου</sup> Εξαμήνου,

Τομέας Λογικού και Υπολογιστών

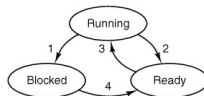
Ιωάννης Χατζηγιαννάκης

Τετάρτη 28 Απριλίου 2010  
Αίθουσα ΒΑ



## Εκτέλεση Διεργασιών

- ▶ Οι διεργασίες είναι ανεξάρτητες οντότητες
- ▶ Οι διεργασίες εκτελούνται σειριακά ανά διαθέσιμο επεξεργαστή
- ▶ Μόνο μια διεργασία είναι ενεργή ανά πάσα στιγμή
- ▶ Επομένως μια διεργασία μπλοκάρει επειδή δεν μπορεί να συνεχίσει την εκτέλεση της
- ▶ Υπάρχουν τέσσερις πιθανοί τρόποι μετάβασης μεταξύ των καταστάσεων μιας διεργασίας:

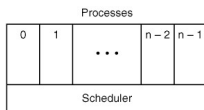


1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available



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



## 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_progrp; /* 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 means catch */
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 handled */
struct sigaction mp_sigact[_NSIG + 1]; /* as in sigaction */
vir_bytes mp_sigreturn; /* address of C library __sigreturn */
struct timer mp_timer; /* watchdog timer for alarm(2)

/* Scheduling priority. */
signed int mp_nice; /* nice is PRIORITY_MIN..PRIORITY_MAX, standard */
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 directory */
    struct inode *fp_rootdir; /* pointer to current root directory */
    struct filp *fp_filp[OPEN_MAX]; /* the file descriptors */
    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't */
int fp_nbytes; /* place to save bytes if rd/wr can't */
int fp_cum_io_partial; /* partial byte count if rd/wr can't */
char fp_suspended; /* set to indicate process hanging */
char fp_revived; /* set to indicate process being revived */
int fp_task; /* which task is proc suspended on */
char fp_sesldr; /* true if proc is a session leader */
char fp_execeed; /* true if proc has exec()ed after fork */
pid_t fp_pid; /* process id */
long fp_cloexec; /* bit map for POSIX Table 6-2 FD_CLOEXEC */
int fp_endpoint; /* kernel endpoint number of this process */
} 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;
}
```

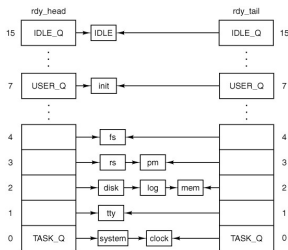


## Χρονοπρογραμματισμός διεργασιών

- ▶ 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
  - ▶ Όταν ολοκληρωθεί το quantum γίνεται block έως ότου ξανά έρθει η σειρά της
  - ▶ Το μέγεθος του quantum είναι διαφορετικό για κάθε κατηγορία διεργασιών



## 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/lowest 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];
```



- ▶ 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;

```



- ▶ 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 επιλεγεί κάποια διεργασία, τότε εκτελείται 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 ποιά 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 */

#ifdef 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();

#ifdef 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 un
#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 ...



## Σύνοψη 10<sup>ης</sup> Διάλεξης

### Προηγούμενο Μάθημα

#### Διεργασίες

Χρονοπρογραμματισμός στα αλληλεπιδραστικά συστήματα  
Υλοποίηση Μηχανισμού Χρονοπρογραμματισμού

### Λειτουργικό Σύστημα Minix

#### Διαχείριση Μνήμης

#### Διαχείριση Ελεύθερης Μνήμης

#### Υλοποίηση Διαχείρισης Μνήμης

### Σύνοψη Μαθήματος

#### Σύνοψη Μαθήματος

#### Τέταρτη Άσκηση

#### Επόμενη Διάλεξη



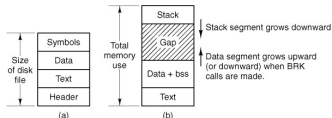
## Θέματα Σχεδιασμού / Αρχιτεκτονικής

- ▶ 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 μετρείται 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 μετριοούνται 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_mmap[NR_LOCAL_SEGS];
```

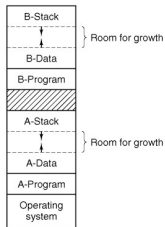
### Πληροφορία στον Διαχειριστή Διεργασιών

```
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 πς 3 βοηθητικές σταθερές (T, D, S)

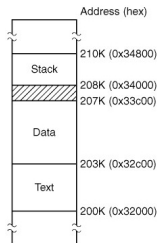


## Τμήματα Διεργασιών – Μέγεθος Τμημάτων



- ▶ 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

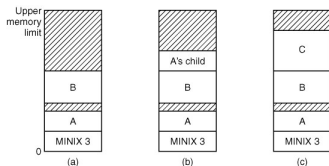
## Τμήματα Διεργασιών – Παράδειγμα



- ▶ 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	0xad	0x2

## Εκτέλεση Εντολών – Δημιουργία Νέων Διεργασιών



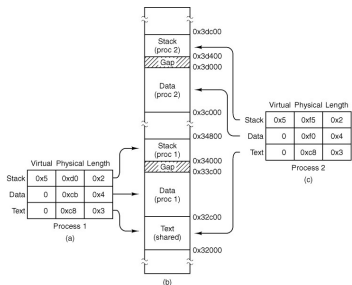
- ▶ Θέλουμε να εκτελέσουμε μια νέα εντολή
- ▶ Αρχικά χρησιμοποιούμε την κλήση του συστήματος fork
  - ▶ Αντιγράφονται τα τμήματα μνήμης της A
- ▶ Αμέσως μετά χρησιμοποιούμε την κλήση του συστήματος exec
  - ▶ Αντικαθίστονται τα τμήματα μνήμης από αυτά της C

## Κοινά Τμήματα Μνήμης

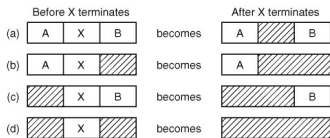
- ▶ Στα αλληλεπιδραστικά συστήματα μια διεργασία μπορεί να 'τρέχει' πολλές φορές
  - ▶ ... αντίγραφα του ίδιου κώδικα στην μνήμη
  - ▶ π.χ. χρησιμοποιώντας την κλήση του συστήματος fork
- ▶ Στην ουσία το τμήμα text είναι κοινό για όλες τις διεργασίες
- ▶ Θέλουμε να εντοπίσουμε κατά πόσο μια διεργασία 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
```

## Κονά Τμήματα Μνήμης – Παράδειγμα

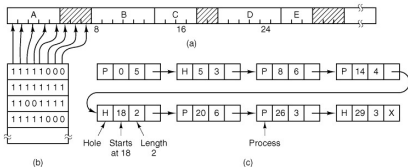


## Ελεύθερη Μνήμη – Οπές Μνήμης (Memory Holes)



- ▶ 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. 1) μελετήσαμε 2 μηχανισμούς διατήρησης των οπών μνήμης – έστω η παραπάνω κατάσταση της μνήμης (α)
  - ▶ 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 free  
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` ανατρέπει 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 swapping 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 removed
{
    /* 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`
  - ▶ Όταν είναι ενεργοποιημένος ο μηχανισμός `swapping`
- ▶ Προσπαθεί να εντοπίσει κάποια υπάρχουσα οπή που είναι 'δίπλα' στην θέση μνήμης που πρόκειται να απελευθερωθεί
  - ▶ Συγκώνευση 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 free
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 impossible
    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);
}
}

```



### Προηγούμενο Μάθημα

#### Διεργασίες

Χρονοπρογραμματισμός στα αλληλεπιδραστικά συστήματα  
Υλοποίηση Μηχανισμού Χρονοπρογραμματισμού

### Λειτουργικό Σύστημα Minix

#### Διαχείριση Μνήμης

#### Διαχείριση Ελεύθερης Μνήμης

Υλοποίηση Διαχείρισης Μνήμης

### Σύνοψη Μαθήματος

#### Σύνοψη Μαθήματος

#### Τέταρτη Άσκηση

Επόμενη Διάλεξη



- ▶ Διαχείριση Μνήμης
- ▶ Λίστα Οπών Μνήμης
- ▶ Μηχανισμοί Διαχείρισης Μνήμης



- ▶ Βιβλίο "Operating Systems Design and Implementation, Third Edition" (Andrew S. Tanenbaum)
  1. Κεφάλαιο 4: Memory Management
    - ▶ Παράγραφος 4.2 Swapping
    - ▶ Παράγραφος 4.7.1 Memory Layout
    - ▶ Παράγραφος 4.7.3 Process Manager Data Structures and Algorithms
    - ▶ Παράγραφος 4.7.4 The FORK, EXIT, and WAIT System Calls
    - ▶ Παράγραφος 4.8.1 The Header Files and Data Structures
    - ▶ Παράγραφος 4.8.2 The Main Program
    - ▶ Παράγραφος 4.8.8 Memory Management Utilities
- ▶ Βιβλίο "Σύγχρονα Λειτουργικά Συστήματα" (A.Tanenbaum)
  1. Κεφάλαιο 4: Διαχείριση Μνήμης
    - ▶ Παράγραφος 4.2 Swapping
    - ▶ Παράγραφος 4.8 Segmentation



- ▶ Αφορά το λειτουργικό σύστημα MINIX 3
- ▶ Οι απαντήσεις πρέπει να ακολουθούν συγκεκριμένη μορφή
- ▶ Παράδοση γίνεται με την χρήση του εργαλείου *submit*
- ▶ Μέχρι **Δευτέρα 10 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<sup>ο</sup> Πρόβλημα

Τροποποιήστε τον αλγόριθμο χρονοπρογραμματισμού του Minix έτσι ώστε:

1. Να υποστηρίζει 24 ουρές προτεραιότητας.
2. Να υποστηρίζει 4 ουρές προτεραιότητας.
3. Να έχει μόνο 1 ουρά προτεραιότητας, δηλαδή να εκφυλιστεί στον αλγόριθμο Round Robin.



## 2<sup>ο</sup> Πρόβλημα

- ▶ Αναπτύξτε μια νέα κλήση του συστήματος (system call) στα πλαίσια του διαχειριστή διεργασιών (process manager)
- ▶ Δέχεται έναν ακέραιο ως παράμετρο
- ▶ Επιστρέφει στην διεργασία που την κάλεσε την εγγραφή του πίνακα διεργασιών που αντιστοιχεί στην παράμετρο (σύμφωνα με το πεδίο process ID).



## 3<sup>ο</sup> Πρόβλημα

- ▶ Αναπτύξτε μια νέα κλήση του συστήματος (system call) στα πλαίσια του διαχειριστή διεργασιών (process manager)
- ▶ Επιστρέφει στην διεργασία που την κάλεσε τα ακόλουθα στοιχεία:
  1. το πλήθος των διεργασιών
  2. τη διεργασία με το μέγιστο χρόνο χρήσης του συστήματος
  3. το συνολικό χρόνο χρήσης του συστήματος
  4. το πλήθος των διεργασιών που είναι σε παύση (PAUSE)
  5. το πλήθος των διεργασιών που είναι σε αναμονή (WAIT)



## Επόμενη Διάλεξη

- ▶ Διαχείριση Σημάτων στο Λ.Σ. MINIX 3
- ▶ Συσκευές Εισόδου / Εξόδου στο Λ.Σ. MINIX 3
- ▶ Επανάληψη από μάθημα 'Λειτουργικά Συστήματα Ι'
  1. Κεφάλαιο 5: Συστήματα Αρχείων
- ▶ Βιβλίο 'Σύγχρονα Λειτουργικά Συστήματα' (A.Tanenbaum)
  1. Κεφάλαιο 5: Συσκευές Εισόδου / Εξόδου
  2. Κεφάλαιο 6: Συστήματα Αρχείων
  3. Κεφάλαιο 10: Μελέτη Περίπτωσης 1: Unix και Linux
    - ▶ Παράγραφος 10.5 Εισόδος / Εξόδος στο UNIX
    - ▶ Παράγραφος 10.6 Το Σύστημα Αρχείων στο UNIX
- ▶ Βιβλίο 'Operating Systems Design and Implementation, Third Edition'' (Andrew S. Tanenbaum)
  1. Κεφάλαιο 3: Input / Output
  2. Κεφάλαιο 5: File System

