Προγραμματισμός Δικτύων – Ε-01

5η Διάλεξη

Διδάσκων: Νίκος Ντάρμος

<ntarmos@cs.uoi.gr>
[http://www.cs.uoi.gr/~ntarmos/Courses/NetworkProgramming/]

Τμήμα Πληροφορικής
Πανεπιστήμιο Ιωαννίνων

Στο σημερινό μάθημα...

- Ταυτοχρονισμός πελάτη και εξυπηρετητή.
- Έλεγχος πρόσβασης, συγχρονισμός.
- Εισαγωγή στην πολυεπεξεργασία βασισμένη σε γεγονότα.

Ταυτοχρονισμός

Εισαγωγή

Εισαγωγή

Ταυτοχρονισμός

- Αφορά την δυνατότητα παράλληλης αποστολής ή/και επεξεργασίας αιτημάτων.
- Υπάρχει σε διάφορα επίπεδα:
  - Επίπεδο δικτύου: υπεύθυνα είναι τα πρωτόκολλα με την πολυπλεξία που παρέχουν.
  - Επίπεδο πελάτη: συνήθως απλά παρέχεται από το λειτουργικό σύστημα.
  - Επίπεδο εξυπηρετητή: σημαντικό κομμάτι του σχεδιασμού και της υλοποίησης μιας εφαρμογής.

Εισαγωγή

Το πρόβλημα C10K

- Σήμερα ο καθένας μπορεί να αγοράσει έναν υπολογιστή με 2 cores @ 2GHz, 2 GBytes RAM και μία κάρτα δικτύου 1 Gbps.
- Με 10000 πελάτες, ένας τέτοιος εξυπηρετητής θα πρέπει να διαθέτει στον κάθε πελάτη 400kHz, 200 kBytes RAM και 100 kbps.

$\Rightarrow$ **Σίγουρα αρκετά για να πάρει ένα αρχείο 4kBytes από το δίσκο και να το στείλει στο δίκτυο, σε καθέναν από τους 10000 πελάτες, μία φορά το δευτερόλεπτο...**

Πως μπορούμε να επιτύχουμε αυτό τον παραλληλισμό/ταυτοχρονισμό;

`http://www.kegel.com/c10k.html`

Εισαγωγή

- Ταυτοχρονισμός μπορεί να επιτευχθεί με:
  - Πολλαπλές διεργασίες (processes).
    - Κάθε διεργασία αποτελεί ακριβές αντίγραφο της αρχικής, με δικό της χώρο διευθύνσεων (address space, heap, stack, registers, κτλ.)
    - Πληρώνουμε το κόστος του context switching.
  - Πολλαπλά νήματα (threads).
    - Κάθε νήμα εκτελεί τον κώδικα μίας «συνάρτησης» και μοιράζεται των χώρο διευθύνσεων με τα υπόλοιπα νήματα της διεργασίας.
    - Θεωρητικά δεν υπάρχει κόστος για context switching· πρακτικά όμως υπάρχει...
  - Σχεδίαση βασισμένη σε συμβάντα (data/event-driven design) και ασύγχρονη Ε/Ε.
    - Ο πελάτης/εξυπηρετητής αποτελείται από μία διεργασία με ένα νήμα εκτέλεσης.
    - Δεν υπάρχει καθόλου context switching.
    - Ενδείκνυται για περιπτώσεις που ο χρόνος εξυπηρέτησης των αιτημάτων είναι μικρός ή στον οποίο επικρατεί το κόστος Ε/Ε.
  - Υβριδικές λύσεις που συνδυάζουν τα παραπάνω.

Διεργασίες

Διεργασίες

- Η διεργασία (process) είναι ένα στιγμιότυπο ενός προγράμματος όταν αυτό εκτελείται.
- Το ίδιο πρόγραμμα μπορεί να εκτελεστεί πολλές φορές ως πολλές διεργασίες.
- Πολλά διαφορετικά προγράμματα μπορούν να εκτελούνται παράλληλα ως πολλές διεργασίες.
- Χρειάζεται μία διεργασία για να δημιουργηθεί μία διεργασία...
- Κάθε διεργασία αποτελείται από:
  - Ένα αναγνωριστικό αριθμό (process ID -- PID).
  - Τον εκτελούμενο κώδικα.
  - Τον χώρο μνήμης στον οποίο έχει πρόσβαση ο κώδικας αυτός.
  - Διάφορες μεταβλητές κατάστασης (program counter, register values, stack pointer, address mappings, TLB, κτλ.)
  - Χειριστές σημάτων (signal handlers).
  - Χειριστές κοινόχρηστων δεδομένων (semaphores, shared memory segments, κτλ.)

Δημιουργία διεργασιών

- Στο Unix διεργασίες δημιουργούνται με την κλήση συστήματος `fork(2)`.
  - `pid_t fork(void);`.
- Μετά την κλήση της `fork(2)` έχουμε δύο διεργασίες:
  - Αυτή που έκανε την κλήση (γονική διεργασία – parent process).
  - Αυτή που δημιουργήθηκε από την κλήση (διεργασία παιδί – child process).
  - Διεργασίες παιδιά από την ίδια γονική διεργασία ονομάζονται «αδέρφια» (siblings).
- Επιστρέφει:
  - 0 στην διεργασία παιδί.
  - Το αναγνωριστικό της διεργασίας παιδιού στην γονική διεργασία.
  - -1 για σφάλμα.
- Η διεργασία παιδί είναι ακριβές αντίγραφο της γονικής διεργασίας.
  - Εκτελεί τον ίδιο κώδικα.
  - Ο program counter είναι ο ίδιος (συνεχίζει από το σημείο αμέσως μετά την `fork(2)`).
  - Παίρνει ένα πλήρες **αντίγραφο** των δεδομένων της γονικής διεργασίας, μαζί με τους file descriptors και τους signal handlers.

Δημιουργία διεργασιών

Παράδειγμα

- Ο κώδικας πρέπει να ελέγχει την τιμή επιστροφής της `fork(2)` για να διαφοροποιείται ανάμεσα σε γονική διεργασία και διεργασία παιδί.

```
1   pid_t pid;
2
3   pid = fork();
4   if (pid < 0) {
5       // Failure
6   } else if (pid == 0) {
7       // Child process
8   } else {
9       // Parent process
10  }
```

- Μία διεργασία μπορεί να μάθει το αναγνωριστικό της μέσω της `getpid(2)` και το αναγνωριστικό της γονικής διεργασίας μέσω της `getppid(2)`.
  - `pid_t getpid(void);`
  - `pid_t getppid(void);`

Δημιουργία διεργασιών

Προσοχή!

Εύκολα χάνεται ο έλεγχος!

Fork Bomb

- Πόσες διεργασίες δημιουργεί ο παρακάτω κώδικας;

```
1  #include <sys/types.h>
2  #include <unistd.h>
3
4  int main() {
5      int i;
6      for (i = 0; i < 10; i ++)
7          fork();
8  }
```

Αντικατάσταση διεργασιών

- Μία διεργασία μπορεί να εκτελέσει ένα πρόγραμμα (και να αντικατασταθεί) μέσω των κλήσεων `exec*(3)`.
  - `int execl(const char* path, const char* arg, ..., NULL);`
  - `int execlp(const char* path, const char* arg, ..., NULL);`
  - `int execle(const char* path, const char* arg, ..., NULL, char *const envp[]);`
  - `int execve(const char* path, const char *const argv[], char *const envp[]);`
  - `int exect(const char *path, char *const argv[], char *const envp[]);`
  - `int execv(const char *path, char *const argv[]);`
  - `int execvp(const char *file, char *const argv[]);`
- Η παράμετρος 0 είναι πάντα το όνομα του προγράμματος (`argv[0]`).
- Η «τελευταία» παράμετρος γραμμής εντολών πρέπει πάντα να είναι `NULL`.

Αντικατάσταση διεργασιών

- Αντικαθίσταται ο κώδικας και ο χώρος μνήμης της αρχικής διεργασίας και η εκτέλεση αρχίζει από το μηδέν.
- Διατηρούνται οι μεταβλητές περιβάλλοντος και οι file descriptors.
- Αν η κλήση επιτύχει, δεν υπάρχει οδός επιστροφής.
- Συνήθως οι `fork(2)` και `exec*(3)` χρησιμοποιούνται μαζί.
  - Π.χ. από το shell για να εκτελέσει τις εντολές.

```
1  if (fork() == 0) {
2      execl("/bin/ls", "ls", "-l", "/home/ntarmos", NULL);
3      perror("execl");
4  }
```

Τερματισμός διεργασιών

- Μια διεργασία τερματίζει όταν:
  1. τελειώσει η εκτέλεση της `main(...)` της ή
  2. καλέσει τη συνάρτηση `exit(3)`.
- Μια διεργασία δε μπορεί να τερματίσει απευθείας μία άλλη διεργασία.
  - Μπορεί μόνο να της στείλει ένα σήμα (signal).
  - Η διεργασία «παραλήπτης» θα εκτελέσει τον αντίστοιχο χειριστή (signal handler).
    - Ίσως τερματίσει, ίσως όμως να κάνει και κάτι άλλο...
    - `SIGINT`: εξ ορισμού τερματίζει, μπορεί όμως να αλλαχθεί (`^C`).
    - `SIGKILL`: εξ ορισμού τερματίζει, δε μπορεί να αλλαχθεί.

Zombies

- Το λειτουργικό σύστημα δεν απελευθερώνει όλους τους πόρους μίας διεργασίας όταν αυτή τερματίσει.
  - Η γονική διεργασία ίσως ενδιαφέρεται για την τιμή επιστροφής της.
- Η διεργασία μετατρέπεται σε «zombie».
  - Η εκτέλεσή της έχει ολοκληρωθεί.
  - Η γονική διεργασία δεν έχει λάβει την τιμή τερματισμού της.
  - Αν η γονική διεργασία έχει ήδη τερματίσει, η διεργασία zombie «υιοθετείται» από την `init()` (PID: 1).
- Διεργασίες που παραμένουν σε κατάσταση zombie καταναλώνουν πόρους συστήματος.
- Η γονική διεργασία απελευθερώνει τους πόρους αυτούς «περιμένοντας» τον τερματισμό της διεργασίας παιδιού.
  - Μέσω των κλήσεων συστήματος `wait*(2)`.

```
pid_t wait(int *status);
pid_t waitpid(pid_t wpid, int *status, int options);
pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t wpid, int *status, int options, struct rusage
*rusage);
```

  - Μέσω ενός χειριστή σήματος για το σήμα `SIGCHLD` (ή αγνόησής του).

Zombies

`wait*(2)`

- Οι κλήσεις αυτές μπλοκάρουν έως ότου κάποιο παιδί αλλάξει κατάσταση (π.χ. τερματίσει) ή επιστρέφουν αμέσως αν υπάρχει κάποιο zombie.
- Η `wait(2)` είναι η απλούστερη:
  - Επιστρέφει το process ID του παιδιού που άλλαξε κατάσταση και θέτει το `*status` στην τιμή τερματισμού του.
- Η `waitpid(2)` δίνει περισσότερο έλεγχο.
  - Η παράμετρος `wpid` καθορίζει για ποιο παιδί θα περιμένουμε (=-1 σημαίνει για οποιοδήποτε).
  - Η παράμετρος `options` καθορίζει διάφορες επιλογές. Π.χ. `WNOHANG` σημαίνει ότι η κλήση δε θα μπλοκάρει αν δεν υπάρχει ήδη κάποιο παιδί σε κατάσταση zombie.

```
1  void my_sigchld(int sig) {
2      while(waitpid(-1, NULL, WNOHANG) > 0) {}
3      signal(SIGCHLD, my_sigchld);
4  }
```

Εισαγωγή

- Κάθε διεργασία είναι χωριστή «οντότητα» και δε μπορεί εν γένει να επηρεάσει άμεσα τις υπόλοιπες διεργασίες του συστήματος.
  - Χωριστός χώρος διευθύνσεων.
  - Χωριστές μεταβλητές περιβάλλοντος και συστήματος.
- Μπορεί ωστόσο να «επικοινωνήσει» μαζί τους, μέσω μηχανισμών inter-process communication (IPC).
  - Μέσω «σημάτων» (signals).
  - Μέσω «σωλήνων» (pipes & fifos).
  - Μέσω «υποδοχών» (sockets).
  - Μέσω «κοινής μνήμης» (shared memory).
  - Μέσω «σημαφόρων» (semaphores).
- Εκτός από τις υποδοχές, οι υπόλοιποι μηχανισμοί λειτουργούν μόνο για διεργασίες στον ίδιο κόμβο.

Σήματα

- Τα σήματα είναι ένας τρόπος να ειδοποιήσουμε μία διεργασία για κάποια κατάσταση ή/και να της ζητήσουμε να κάνει κάποια λειτουργία.
- Σε κάθε δυνατό σήμα αντιστοιχεί και μία συνάρτηση «χειριστής» (signal handler).
- Παραδίδονται και επεξεργάζονται **ασύγχρονα**.
  - Όταν μια διεργασία λάβει ένα σήμα **διακόπτει** την κανονική ροή εκτέλεσής της και εκτελεί τον χειριστή του σήματος.
  - **Προσοχή:** σήματα μπορεί να έρθουν ανά πάσα στιγμή!
- Μία διεργασία μπορεί να στείλει ένα σήμα σε μία άλλη μέσω της κλήσης συστήματος `kill(2)`:

```
int kill(pid_t pid, int sig);
```

  - `pid`: το αναγνωριστικό της διεργασίας στην οποία θα σταλεί το σήμα.
    0: το σήμα στέλνεται σε όλους τις διεργασίες τις ομάδας διεργασιών.
    -1: το σήμα στέλνεται σε όλες τις διεργασίες στις οποίες έχει πρόσβαση ο χρήστης.
  - `sig`: ο αριθμός του σήματος που θα σταλεί.
    0: έλεγχος ορθότητας παραμέτρων.

Σήματα

- Σήματα για καταστάσεις σφαλμάτων:
  - `SIGFPE`: λάθος πράξης κινητής υποδιαστολής.
  - `SIGSEGV`: πρόσβαση σε θέση μνήμης που δεν ανήκει στη διεργασία.
  - `SIGBUS`: πρόσβαση σε θέση μνήμης με λάθος ευθυγράμμιση (alignment).
  - `SIGPIPE`: εγγραφή σε pipe χωρίς αναγνώστη.
- Σήματα για έλεγχο εκτέλεσης διεργασιών:
  - `SIGINT`: διακοπή εκτέλεσης.
  - `SIGTERM`: «ομαλός» τερματισμός διεργασίας.
  - `SIGKILL`: τερματισμός διεργασίας.
  - `SIGSTOP/SIGTSTP, SIGCONT`: παύση κι επανεκκίνηση εκτέλεσης.
  - `SIGCHLD`: αλλαγή κατάστασης διεργασίας παιδιού.
- Σήματα για «ενημέρωση» διεργασιών:
  - `SIGALRM`: «ξυπνητήρι».
  - `SIGUSR1, SIGUSR2`: σήματα γενικής χρήσης.
- Για πλήρη λίστα δείτε την `signal(3)` και εκτελέστε την εντολή `kill -l`.

Σήματα

- Ο χειριστής είναι απλά μία συνάρτηση `void func(int)`.
- Εγκαθίσταται με τις συναρτήσεις `signal(3)` και `sigaction(2)`:
  ```
  void (*signal(int sig, void (*func)(int)))(int);
  int sigaction(int sig, const struct sigaction* act, struct sigaction *oact);
  ```
- Ορίζονται οι «συναρτήσεις» `SIG_IGN` (αγνόησε το σήμα) και `SIG_DFL` (προκαθορισμένος χειριστής από το σύστημα).
- Αν χρησιμοποιούμε την `signal(3)`, ο χειριστής επαναρχικοποιείται στον προκαθορισμένο από το σύστημα σε κάθε κλήση του και πρέπει να το προβλέπουμε στο χειριστή.

Σήματα

Παράδειγμα

```
1   #include <stdio.h>
2   #include <sys/types.h>
3   #include <signal.h>
4
5   void my_sigusr1(int sig) {
6       printf("Got signal #%d\n", sig);
7       signal(sig, my_sigusr1);
8   }
9   void my_sigchld(int sig) {
10      while(waitpid(-1, NULL, WNOHANG) > 0) {}
11      signal(SIGCHLD, my_sigchld);
12  }
13
14  int main() {
15      void (*oldhandler)(int);
16      oldhandler = signal(SIGUSR1, my_sigusr1);
17      signal(SIGCHLD, my_sigchld);
18      signal(SIGCHLD, SIG_IGN);
19      // ...
20      kill(getpid(), SIGUSR1);
21      // ...
22  }
```

Σήματα

Προσοχή!

- Οι χειριστές εκτελούνται ασύγχρονα!
- Η ροή του προγράμματός μας ίσως διακοπεί από λήψη σήματος!
- Δε μπορούμε να χρησιμοποιήσουμε όλες τις συναρτήσεις μέσα σε έναν χειριστή!
- Δεν υπάρχει «ουρά σημάτων»!
- Προσοχή με τις κοινές/καθολικές μεταβλητές (π.χ. `errno`)!

Pipes & FIFOs

- Οι «σωλήνες» (pipes) είναι μονόδρομα κανάλια επικοινωνίας ανάμεσα σε δύο διεργασίες.
  - Ό,τι γράφεται στο ένα άκρο, διαβάζεται από το άλλο.
  - Αν απαιτείται αμφίδρομη επικοινωνία, χρησιμοποιούμε είτε 2 pipes είτε sockets (`socketpair(2)`).
- Δημιουργούνται με χρήση της συνάρτησης `pipe(2)`.
  `int pipe(int *fildes);`
  - Το `fildes` είναι πίνακας 2 θέσεων:
    - `fildes[0]`: προς ανάγνωση.
    - `fildes[1]`: προς εγγραφή.
- Μόνο διεργασίες με κοινή γονική διεργασία μπορούν να επικοινωνήσουν μέσω «σωλήνων». Για διαφορετικές περιπτώσεις, υπάρχουν τα FIFOs (ή named pipes).
  - Καθορίζεται ένα ειδικό αρχείο μέσω του οποίου γίνεται η επικοινωνία.
  - Το αρχείο δημιουργείται με την `mkfifo(2)`.
    - `int mkfifo(const char *path, mode_t mode);`
  - Κατόπιν το χειριζόμαστε ως αρχείο (`open(2)`, `read(2)`, `write(2)`, `close(2)`).

Pipes & FIFOs

Παράδειγμα

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main() {
5       int pid, fd[2];
6       char buf[14];
7
8       if (pipe(fd) < 0) return -1;
9       if ((pid = fork()) == 0) {
10          close(fd[0]);
11          write(fd[1], "Hello father!", 14);
12      } else {
13          close(fd[1]);
14          read(fd[0], buf, 14);
15          printf("Child said: %s\n", buf);
16          waitpid(pid, NULL, 0);
17      }
18      return 0;
19  }
```

Κοινή μνήμη

- Οι διεργασίες έχουν καθεμία το δικό της χώρο διευθύνσεων.
- Ο μηχανισμός της «κοινής μνήμης» (shared memory) επιτρέπει σε πολλές διεργασίες να έχουν ταυτόχρονα πρόσβαση σε ένα τμήμα μνήμης.
- Η κοινή μνήμη οργανώνεται σε τμήματα (segments).
- Τα τμήματα αυτά πρέπει:
  1. Να δημιουργούνται (`shmget(2)`).
  2. Να προσαρτώνται από τις αντίστοιχες διεργασίες (`shmat(2)`).
  3. Μετά τη χρήση τους, να αποπροσαρτώνται (`shmdt(2)`).
  4. Η τελευταία διεργασία πρέπει να τα καταστρέφει (`shmctl(2)`).
- **Προσοχή: Κοινή πρόσβαση $\Rightarrow$ race conditions.**

Κοινή μνήμη

Δημιουργία: `shmget(2)`

- `int shmget(key_t key, int size, int flag);`
  - `key`: αναγνωριστικό με το οποίο ταυτοποιείται το νέο τμήμα κοινής μνήμης.
    - `IPC_PRIVATE`: προς χρήση μόνο από διεργασίες παιδιά της παρούσας διεργασίας.
    - Για τη δημιουργία του αναγνωριστικού μπορεί να χρησιμοποιηθεί η συνάρτηση `ftok(3)`.
      `key_t ftok(const char *path, int id);`
  - `size`: μέγεθος (σε bytes) του τμήματος.
  - `flag`:
    - `IPC_CREAT` για δημιουργία νέου τμήματος.
    - `IPC_EXCL` για έλεγχο μοναδικότητας του νέου τμήματος.
    - Δικαιώματα πρόσβασης στο νέο τμήμα (`rw-rw-rw-`).
  - Επιστρέφει το αναγνωριστικό του τμήματος ή -1 για σφάλμα.
- Παράδειγμα:

```
1  #include <sys/ipc.h>
2  #include <sys/shm.h>
3  key_t shmkey;
4  int shmid;
5  shmkey = ftok(argv[0], getpid());
6  if (shmkey == -1) error(/* ... */);
7  shmid = shmget(shmkey, 10, IPC_CREAT | IPC_EXCL | 0640);
8  if (shmid == -1) error(/* ... */);
```

Κοινή μνήμη

Προσάρτηση: `shmat(2)`

- `void* shmat(int shmid, const void *addr, int flag);`
  - `shmid`: αναγνωριστικό του τμήματος που θα προσαρτηθεί, όπως επιστράφηκε από την `shmget(2)`.
  - `addr`: διεύθυνση στην οποία θα προσαρτηθεί το τμήμα.
    - `0`: αποφασίζει ο πυρήνας του λειτουργικού.
    - `!0` και `!(flag & SHM_RND)`: στην διεύθυνση `addr`.
    - `!0` και `flag & SHM_RND`: στην κοντινότερη «ευθυγραμμισμένη» διεύθυνση στο `addr`.
  - `flag`: διάφορες επιλογές.
    - `SHM_RND`: όπως παραπάνω.
    - `SHM_RDONLY`: πρόσβαση μόνο ανάγνωσης.
  - Επιστρέφει δείκτη στη διεύθυνση στην οποία έγινε η προσάρτηση ή -1 για σφάλμα.

Αποπροσάρτηση: `shmdt(2)`

- `int shmdt(const void *addr);`
  - `addr`: δείκτης στο προς αποπροσάρτηση τμήμα.
  - Επιστρέφει 0 για επιτυχία, -1 για αποτυχία.
  - **Προσοχή: Δεν καταστρέφει το τμήμα!**
    Τα τμήματα δεν καταστρέφονται όταν τερματίσουν οι αντίστοιχες διεργασίες!

Κοινή μνήμη

Έλεγχος και καταστροφή: `shmctl(2)`

- `int shmctl(int shmid, int cmd, struct shmid_ds *buf);`
  - `shmid`: αναγνωριστικό του τμήματος που θα επεξεργαστεί η συνάρτηση.
  - `cmd`:
    - `IPC_STAT`: αποθηκεύει πληροφορίες για το τμήμα στο χώρο που δείχνει ο δείκτης `buf`.
    - `IPC_SET`: αλλάζει τον ιδιοκτήτη (`uid` και `gid`) και τα δικαιώματα πρόσβασης (`mode`) του τμήματος σε αυτά που καθορίζει η δομή στην οποία δείχνει ο `buf`.
    - `IPC_RMID`: σημειώνει το τμήμα ως προς καταστροφή. Καμία διεργασία δε μπορεί πλέον να το προσαρτήσει. Όταν το τμήμα δε θα είναι πια προσαρτημένο σε καμία διεργασία, ο πυρήνας θα το απελευθερώσει.
- Δείτε τις `ipcs(1)` και `ipcrm(1)`.

```
struct shmid_ds {
    struct ipc_perm shm_perm;     /* permissions */
    int             shm_segsz;    /* size of segment */
    pid_t           shm_lpid;     /* PID of last op */
    pid_t           shm_cpid;     /* PID of creator */
    short           shm_nattch;   /* # of attached procs */
    time_t          shm_atime;    /* last shmat(2) */
    time_t          shm_dtime;    /* last shmdt(2) */
    time_t          shm_ctime;    /* last shmctl(2) */
    void*           shm_internal; /* SysV compatibility */
};
```

```
struct ipc_perm {
    unsigned short cuid; /* creator uid */
    unsigned short cgid; /* creator gid */
    unsigned short uid;  /* user id */
    unsigned short gid;  /* group id */
    unsigned short mode; /* r/w perms */
    unsigned short seq;  /* sequence # */
    key_t          key;  /* ipc key */
};
```

Κοινή μνήμη

Παράδειγμα

```
1  int main() {
2      int shmid, *intPtr;
3      if ((shmid = shmget(IPC_PRIVATE, sizeof(int), 0600)) == -1)
4          return 1;
5      if ((intPtr = (int*)shmat(shmid, 0, 0)) == (void*)-1)
6          return 2;;
7      *intPtr = 10;
8
9      if (!fork()) { // Child process
10         printf("Shared int value: %d\n", *intPtr);
11         *intPtr = 20;
12         shmdt((void*)intPtr);
13     } else { // Parent process
14         sleep(1);
15         printf("Shared int value: %d\n", *intPtr);
16         shmdt((void*)intPtr);
17         shmctl(shmid, IPC_RMID, 0);
18     }
19     return 0;
20 }
```

Ανταγωνισμός

Κοινή πρόσβαση ⇒ Ανταγωνισμός ⇒ Ανάγκη για συγχρονισμό!

```
1  int *accountBalance; /* shared memory data */
2  if (withdrawal <= *accountBalance)
3      *accountBalance -= withdrawal;
```

Διεργασία 1

```
// account balance = 12000,
// withdrawal = 10000

if (withdrawal <= *accountBalance)
                    Context switch →

    *accountBalance -= withdrawal;

// account balance = -3000,
```

Διεργασία 2

```
// account balance = 12000,
// withdrawal = 5000


if (withdrawal <= *accountBalance)
    *accountBalance -= withdrawal;
←  Context switch

// account balance = -3000,
```

Σημαφόροι

- Η «κλασσική μονάδα» συγχρονισμού.
- Παίρνουν τιμές θετικών ακεραίων.
- Δύο βασικές **ατομικές** (atomic) λειτουργίες:
  - `DOWN(...)`:
    - Αν η τιμή του σημαφόρου είναι ≥ 1 τότε μείωσέ την κατά 1.
    - Αλλιώς (=0) μπλόκαρε τη διεργασία.
  - `UP(...)`:
    - Αν υπάρχουν διεργασίες που έχουν μπλοκάρει στο σημαφόρο, ξεμπλόκαρε μία.
    - Αλλιώς αύξησε την τιμή του σημαφόρου κατά 1.

Σημαφόροι

- Χρησιμοποιούνται είτε για αμοιβαίο αποκλεισμό (mutual exclusion → mutex) είτε για αναμονή κάποιου γεγονότος.
  - Mutex:
    - Θέτουμε το σημαφόρο στην τιμή 1.
      → `DOWN(...)` για να εισέλθουμε στην περιοχή αμοιβαίου αποκλεισμού.
      → `UP(...)` για να εξέλθουμε από την περιοχή αμοιβαίου αποκλεισμού.
  - Αναμονή γεγονότος:
    - Θέτουμε το σημαφόρο στην τιμή 0.
      → `DOWN(...)` για να περιμένουμε για το γεγονός.
      → `UP(...)` για να σηματοδοτήσουμε ότι το γεγονός συνέβη.

```
1  int *accountBalance; /* shared memory data */
2  semaphore mutex = 1;
3  // ...
4  DOWN(mutex);
5  if (withdrawal <= *accountBalance)
6      *accountBalance -= withdrawal;
7  UP(mutex);
```

Σημαφόροι

Δημιουργία: `semget(2)`

- `int semget(key_t key, int nsems, int flag);`
  - Δημιουργεί **πίνακα** από σημαφόρους.
  - `key`: αναγνωριστικό με το οποίο ταυτοποιείται ο πίνακας των σημαφόρων (όπως ακριβώς και στην `shmget(2)`).
  - `nsems`: πλήθος σημαφόρων στον πίνακα.
  - `flag`: όπως ακριβώς και στην `shmget(2)`.
  - Επιστρέφει το αναγνωριστικό του πίνακα των σημαφόρων ή -1 για σφάλμα.

Σημαφόροι

Πράξεις: `semop(2)`

- `int semop(int semid, struct sembuf *array, size_t nops);`
  - Εκτελεί **ατομικά** τις πράξεις που ορίζονται.
  - `semid`: αναγνωριστικό του πίνακα σημαφόρων στον οποίο θα γίνουν οι πράξεις.
  - `array`: πίνακας από πράξεις.
    - `sem_num`: η θέση του σημαφόρου στον πίνακα.
    - `sem_op`: κατά πόσο να μεταβληθεί η τιμή του σημαφόρου (1: `UP`, -1: `DOWN`, 0: περίμενε έως ότου η τιμή του σημαφόρου γίνει 0).
    - `sem_flg`: καθορίζει διάφορες επιλογές της πράξης (δείτε το man page).
  - `nops`: πλήθος πράξεων στον πίνακα `array`.
  - Επιστρέφει 0 για επιτυχία ή -1 για σφάλμα.

```
void DOWN(int semid, int sem) {
    struct sembuf op;
    op.sem_num = sem;
    op.sem_op = -1;
    op.sem_flg = 0;
    semop(semid, &op, 1);
}
```

```
void UP(int semid, int sem) {
    struct sembuf op;
    op.sem_num = sem;
    op.sem_op = 1;
    op.sem_flg = 0;
    semop(semid, &op, 1);
}
```

```
struct sembuf {
    u_short sem_num;
    short   sem_op;
    short   sem_flg;
};
```

Κοινή μνήμη

Έλεγχος και καταστροφή: `semctl(2)`

- `int semctl(int semid, int semnum, int cmd, ...);`
  - `semid`: αναγνωριστικό του πίνακα σημαφόρου.
  - `semnum`: θέση του σημαφόρου στον πίνακα.
  - `...`: κάποιες πράξεις δέχονται και ως επιπλέον όρισμα έναν union τύπου `semun`.
  - `cmd`:
    - `IPC_STAT`: αποθηκεύει πληροφορίες για τον πίνακα σημαφόρων στο χώρο που δείχνει ο δείκτης `buf`.
    - `IPC_SET`: αλλάζει τον ιδιοκτήτη (`uid` και `gid`) και τα δικαιώματα πρόσβασης (`mode`) του πίνακα σημαφόρων σε αυτά που καθορίζει ο `buf`.
    - `IPC_RMID`: αφαιρεί άμεσα τον πίνακα σημαφόρων από το σύστημα, ξεμπλοκάροντας και όσες διεργασίες περίμεναν σε κάποιον σημαφόρο.
    - `GETVAL`: επέστρεψε την τιμή του σημαφόρου.
    - `SETVAL`: θέσε το σημαφόρο στην τιμή `val`.
    - `GETALL`, `SETALL`: αποθήκευσε ή θέσε τις τιμές όλων των σημαφόρων σε ή ίσες με τον πίνακα `array`.
    - Δείτε το man page...

```
union semun {
    int               val;
    struct semid_ds*  buf;
    u_short*          array;
};
```

```
struct semid_ds {
    struct ipc_perm
                      sem_perm;
    struct sem*       sem_base;
    u_short           sem_nsems;
    time_t            sem_otime;
    long              sem_pad1;
    time_t            sem_ctime;
    long              sem_pad2;
    long              sem_pad3[4];
};
```

Σημαφόροι

Παράδειγμα

```
1  int main() {
2      int my_sem = semget(IPC_PRIVATE, 1, 0600); // Create, value = 0
3      if (!fork()) { // Child process
4          DOWN(my_sem, 0);
5          printf("Child got the mutex\n");
6      } else { // Parent process
7          printf("Parent sleeping for 5'\n");
8          sleep(5);
9          printf("Parent releasing the mutex\n");
10         UP(my_sem, 0);
11         wait(0);
12         semctl(my_sem, 0, IPC_RMID);
13     }
14     return 0;
15 }
```

Νήματα

Εισαγωγή

- Τα πολυ-διεργασιακά συστήματα είναι συνήθως «βαριά».
  - Η `fork(2)` πρέπει να δημιουργήσει αντίγραφα της διεργασίας, του χώρου μνήμης της, κτλ.
  - Χρόνος σπαταλιέται στο context switching.
  - Η επικοινωνία ανάμεσα στις διεργασίες είναι «στρυφνή».
- Η απάντηση: Νήματα (threads).
  - Ουσιαστικά «ελαφρές διεργασίες».
  - Κάθε διεργασία μπορεί να περιέχει πολλά νήματα εκτέλεσης.
  - Κάθε νήμα εκτελεί μία συνάρτηση του συνολικού προγράμματος $\Rightarrow$ Δεν χρειάζεται δημιουργία αντιγράφου του κώδικα της διεργασίας.
  - Όλα τα νήματα μοιράζονται το χώρο μνήμης της διεργασίας $\Rightarrow$ Δεν χρειάζεται δημιουργία αντιγράφου των δεδομένων της διεργασίας.
  - Κάθε νήμα έχει δικά του αντίγραφα των program counter, stack & stack pointer, errno, signal mask και ένα μοναδικό αναγνωριστικό (thread id).
- **Προσοχή:**
  - **Ο παραλληλισμός και η κοινή πρόσβαση των νημάτων στους πόρους της διεργασίας καθιστά υποχρεωτικό τον συγχρονισμό τους!**
  - **Δεν είναι όλες οι κλήσεις συστήματος και οι συναρτήσεις βιβλιοθηκών ασφαλείς για χρήση σε νήματα!**

Εισαγωγή

- Το πρότυπο POSIX ορίζει ένα βασικό API για υποστήριξη νημάτων (POSIX threads ή pthreads).
- Διάφορα μοντέλα υλοποίησης νημάτων:
  - N προς 1: Τα νήματα υλοποιούνται εξολοκλήρου στο επίπεδο χρήστη (user-space) και ο πυρήνας του λειτουργικού συστήματος δεν γνωρίζει και δεν υποστηρίζει τίποτα σχετικό.
  - N προς N: Κάθε νήμα υλοποιείται ως μια διεργασία· νήματα της ίδιας ιδεατής διεργασίας χρησιμοποιούν κοινή μνήμη και ο πυρήνας φροντίζει ώστε να λειτουργούν σαν πραγματικά νήματα.
  - N προς M: Ενδιάμεση λύση όπου ο πυρήνας προσφέρει κάποια υποστήριξη αλλά οι αποφάσεις χρονοδρομολόγησης λαμβάνονται σε χώρο χρήστη.
  - 1 προς 1: «Πραγματικά» νήματα.
- Θεωρητικά δεν υπάρχει context switching. Πρακτικά όμως;

Δημιουργία νημάτων: `pthread_create(3)`

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);`
  - Δημιουργεί ένα νέο νήμα με αναγνωριστικό `thread` και χαρακτηριστικά `attr` (με `NULL` χρησιμοποιεί τα defaults), το οποίο θα εκτελεί τη συνάρτηση `start_routine` με όρισμα `arg`.
- `int pthread_attr_init(pthread_attr_t *attr);`
  - Αρχικοποιεί τη δομή `attr`.
- `int pthread_attr_destroy(pthread_attr_t *attr);`
  - Απελευθερώνει τη δομή `attr`.
- Δείτε το man page του `pthread_attr(3)`.

Τερματισμός νημάτων

- Ένα νήμα τερματίζει όταν:
  - Τερματίσει η διεργασία στην οποία ανήκει.
  - Τερματίσει το γονικό του νήμα.
  - Επιστρέψει η συνάρτηση `start_routine`.
  - Καλέσει τη συνάρτηση `pthread_exit(3)`.
- `void pthread_exit(void *value_ptr);`
  - `value_ptr`: δείκτης στην τιμή επιστροφής του νήματος.
- **Προσοχή:** μη χρησιμοποιείτε διευθύνσεις τοπικών μεταβλητών της `start_routine`!
- `void pthread_join(pthread_t thread, void **value_ptr);`
  - Όπως και με τις διεργασίες, το γονικό νήμα πρέπει να συλλέξει την τιμή επιστροφής του νήματος.
  - Το γονικό νήμα μπλοκάρει έως ότου το νήμα `thread` τερματίσει.
  - Αν το `value_ptr` δεν είναι `NULL`, αποθηκεύει εκεί την τιμή επιστροφής του νήματος.

Τερματισμός νημάτων

- `int pthread_detach(pthread_t thread);`
  - «Αποδεσμεύει» το νήμα `thread`.
  - Ένα νήμα σε αυτή την κατάσταση δε χρειάζεται να γίνει `pthread_join(3)`.
  - Ένα νήμα σε αυτή την κατάσταση δεν τερματίζει όταν τερματίσει το πατρικό του νήμα.
- Όλα τα νήματα είναι εξ ορισμού σε κατάσταση «joinable» (δηλ. δεσμευμένα).
  - Μπορούμε να δημιουργήσουμε αποδεσμευμένο εξ αρχής νήμα χρησιμοποιώντας τη συνάρτηση `pthread_attr_setdetachstate(3)`.

```
1  #include <pthread.h>
2
3  pthread_t id
4  pthread_attr_t attr;
5
6  pthread_attr_init(&attr);
7  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
8  pthread_create(&id, &attr, start_routine, NULL);
```

Νήματα

Παράδειγμα

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  int global_int = 0;
5
6  void* my_thread(void *param) {
7      global_int += *((int*)param);
8      return NULL;
9  }
10
11 int main() {
12     pthread_t id;
13     int p = 10;
14
15     printf("global_int: %d\n", global_int);
16     pthread_create(&id, NULL, my_thread, (void*)&p);
17     pthread_join(id, NULL);
18     printf("global_int: %d\n", global_int);
19     return 0;
20 }
```

Συγχρονισμός νημάτων

- Τα νήματα εξ ορισμού μοιράζονται τους περισσότερους από τους πόρους του γονικού νήματος/διεργασίας.
- **Απαιτείται συγχρονισμός στις προσβάσεις στα κοινόχρηστα στοιχεία!**
- Τα νήματα διαθέτουν δύο μηχανισμούς συγχρονισμού:
  - Σημαφόρους αμοιβαίου αποκλεισμού (mutexes).
  - Μεταβλητές κατάστασης (condition variables).

Συγχρονισμός με mutexes

- Το API των POSIX threads ορίζει τις εξής συναρτήσεις για συγχρονισμό αμοιβαίου αποκλεισμού:

```
1  int pthread_mutexattr_init(pthread_mutexattr_t *attr);
2  int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
3  int pthread_mutex_init(pthread_mutex_t *mutex, const
       pthread_mutexattr_t *attr);
4  int pthread_mutex_lock(pthread_mutex_t *mutex));
5  int pthread_mutex_trylock(pthread_mutex_t *mutex);
6  int pthread_mutex_unlock(pthread_mutex_t *mutex);
7  int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Συναρτήσεις οι οποίες δεν είναι thread-safe μπορούν να χρησιμοποιηθούν σε νήματα αν προστατεύονται από ένα mutex.

Συγχρονισμός με mutexes

Παράδειγμα

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  int accountBalance = 12000;
5  pthread_mutex_t accountMutex;
6  void* my_thread(void *param) {
7      int transaction = *((int*)param);
8      pthread_mutex_lock(&accountMutex);
9      if (transaction >= 0 || -transaction < accountBalance)
10         accountBalance += transaction;
11     pthread_mutex_unlock(&accountMutex);
12     return NULL;
13 }
14
15 int main() {
16     pthread_t thread1, thread2;
17     int transaction1= -10000, transaction2 = -5000;
18     pthread_mutex_init(&accountMutex, NULL);
19     pthread_create(&thread1, NULL, my_thread, (void*)&transaction1);
20     pthread_create(&thread2, NULL, my_thread, (void*)&transaction2);
21     pthread_detach(thread1);
22     pthread_detach(thread2);
23     sleep(2);
24     printf("Account balance: %d\n", accountBalance);
25     pthread_mutex_destroy(&accountMutex);
26     return 0;
27 }
```

Συγχρονισμός με condition variables

- Πρόβλημα: Τι γίνεται αν τα mutexes δεν αρκούν για να λύσουν το πρόβλημα;
  - Π.χ. αν κάποια διεργασία πρέπει να περιμένει κάποιο γεγονός, ενώ είναι σε τμήμα προστατευόμενο από mutex;
- Απάντηση: Μεταβλητές Κατάστασης.
  - Οι μεταβλητές κατάστασης πάντα συνυπάρχουν με ένα mutex και μία μεταβλητή ελέγχου.
    - Το mutex συγχρονίζει την πρόσβαση στη μεταβλητή κατάστασης και τη μεταβλητή ελέγχου.
    - Η μεταβλητή κατάστασης χρησιμοποιείται για να εγγυηθούμε ότι το γεγονός το οποίο περιμένουμε όντως συνέβη.

```
1  int pthread_condattr_init(pthread_condattr_t *attr);
2  int pthread_condattr_destroy(pthread_condattr_t *attr);
3  int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t
       *attr);
4  int phtread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
5  int phtread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *
       mutex, const struct timespec *abstime);
6  int phtread_cond_signal(pthread_cond_t *cond);
7  int phtread_cond_broadcast(pthread_cond_t *cond);
8  int pthread_cond_destroy(pthread_cond_t *cond);
```

Συγχρονισμός με condition variables

- Αρχικοποίηση:

```
1  pthread_mutex_t mutex;
2  pthread_cond_t  cond;
3  int             accountBalance, withdrawal, deposit;
4  pthread_mutex_init(&mutex, NULL);
5  pthread_cond_init(&cond, NULL);
```

- Αναμονή γεγονότος:

```
1  pthread_mutex_lock(&mutex);
2  while (accountBalance < withdrawal)
3      pthread_cond_wait(&cond, &mutex);
4  accountBalance -= withdrawal;
5  pthread_mutex_unlock(&mutex);
```

- Σηματοδότηση γεγονότος:

```
1  pthread_mutex_lock(&mutex);
2  accountBalance += deposit;
3  pthread_cond_signal(&cond);
4  pthread_mutex_unlock(&mutex);
```

Εισαγωγή σε πολυεπεξεργασία βασισμένη σε γεγονότα

Πολυπλεξία στην Ε/Ε

- Πρόβλημα: Πως μπορούμε να χειριστούμε πολλαπλούς file descriptors ταυτόχρονα;
  - Με πολλές διεργασίες ή/και νήματα.
    ⇒ «Βαριά» λύση.
  - Με non-blocking Ε/Ε.
    ⇒ Δουλεύει καλά για `read(2)` και `write(2)`. Τι γίνεται όμως με την `accept(2);`
- Λύση: `select(2)`.
  - `int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);`
  - Περιμένει το πολύ `timeout` χρόνο (ή για πάντα αν `NULL`) για να γίνει κάποια αλλαγή στους πρώτους `nfds` file descriptors που βρίσκονται στα σύνολα `readfds`, `writefds` και `exceptfds`.
    - `nfds`: ο μεγαλύτερος file descriptor στα σύνολα, συν 1.
    - `readfds`: file descriptors από τους οποίους θα διαβάσουμε δεδομένα ή στους οποίους θα δεχτούμε συνδέσεις.
    - `writefds`: file descriptors στους οποίους θα γράψουμε δεδομένα.
    - `exceptfds`: file descriptors στους οποίους περιμένουμε «επείγοντα δεδομένα» (OOB) ή γενικά ειδικές καταστάσεις.
    - Επιστρέφει το πλήθος των file descriptors που είναι έτοιμοι για επεξεργασία, 0 για τέλος χρόνου, -1 για σφάλμα.

Πολυπλεξία στην Ε/Ε

- Για τη διαχείριση των συνόλων των file descriptors ορίζονται οι εξής μακροεντολές:
  - `FD_SET(fd, &fdset);`
  - `FD_CLR(fd, &fdset);`
  - `FD_ISSET(fd, &fdset);`
  - `FD_ZERO(&fdset);`
- Όταν η `select(2)` επιστρέψει, στα σύνολα αυτά θα είναι «set» μόνο οι file descriptors που είναι έτοιμοι για Ε/Ε.

Πολυπλεξία στην Ε/Ε

Παράδειγμα

```
1  int sd, cd, sel;
2  fd_set fds;
3
4  sd = socket(/* ... */);
5  bind(sd, /* ... */);
6  listen(sd, /* ... */);
7
8  while (1) {
9      FD_ZERO(&fds);
10     FD_SET(sd, &fds);
11     sel = select(sd + 1, &fds, NULL, NULL, NULL);
12     if (sel <= 0) { /* do something */ }
13     if (sel > 0) {
14         if (FD_ISSET(sd, &fds)) {
15             cd = accept(sd, /* ... */);
16             if (cd < 0) error(/* ... */);
17             // Serve the request
18             close(cd);
19         }
20     }
21 }
```

Στο επόμενο μάθημα...

- Πολυεπεξεργασία βασισμένη σε γεγονότα (συνέχεια).