ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS

Λειτουργικά Συστήματα

Διεργασίες
&
Inter-Process Communication

Unix Processes

Programs vs. Processes

- Οι υπολογιστές εκτελούν **προγράμματα**
- Μία διεργασία (**process**) είναι ένα **instance** ενός προγράμματος, καθώς αυτό εκτελείται.
  - Το **ίδιο πρόγραμμα** μπορεί να εκτελείται **πολλές φορές** ταυτόχρονα
  - **Διαφορετικά προγράμματα** μπορούν επίσης να εκτελούνται ταυτόχρονα
  - Ακόμα και από **διαφορετικούς χρήστες**
- Μια διεργασία μπορεί να «γεννηθεί» μόνο από μια άλλη διεργασία
  - Π.χ., όταν πληκτρολογώ μια εντολή στο τερματικό μου (στο shell), το Unix δημιουργεί μια νέα διεργασία για να την εκτελέσει

Στα ενδότερα μιας διεργασίας

- Μια διεργασία αποτελείται από:
  - Το πρόγραμμα που εκτελεί (i.e., a file containing the code to run)
  - **PID**: Process identifier (an integer)
  - **Memory** used to execute the program (text, data, heap, stack)
  - **PC**: Program counter
    - indicates where in the program the process currently is
  - A number of **signal handlers**
    - tells the program what to do when receiving signals

- Μια διεργασία μπορεί να «μάθει» το PID της:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
```

- Ή το PID του γονέα της:

```
pid_t getppid(void);
```

The fork() System Call [1/3]

- Σε συστήματα Unix, υπάρχει ακριβώς **ένας και μόνο ένας τρόπος** να φτιάξεις μια νέα διεργασία:

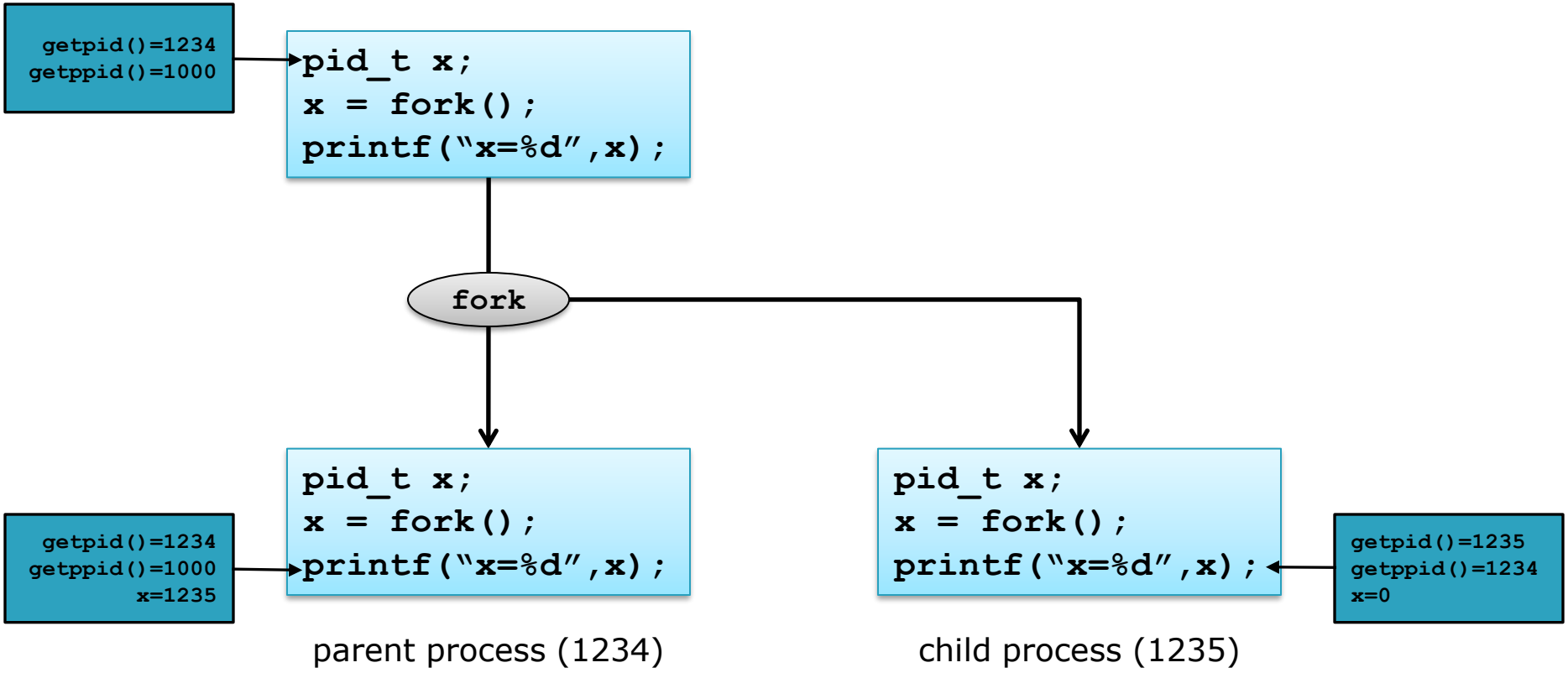
```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

- Η διεργασία παιδί (child process) είναι ακριβές αντίγραφο του γονέα:
  - Τρέχει το **ίδιο πρόγραμμα**
  - Η **μνήμη** της είναι ακριβές αντίγραφο της μνήμης του γονέα
  - Τα **signal handlers** και **file descriptors** αντιγράφονται επίσης
  - Ο **program counter (PC)** είναι ακριβώς στο ίδιο σημείο όπως του γονέα
    - Δηλ., ακριβώς μετά την fork()

- Υπάρχει **ένας και μόνο ένας τρόπος** να διακρίνουμε γονέα από παιδί:
  - **`fork()`** returns 0 to the child process
  - **`fork()`** returns the child's PID to the parent process
  - **`fork()`** returns -1 in case of error

- Σχεδόν πάντα ελέγχουμε το αποτέλεσμα της **`fork()`**.

The fork() System Call [2/3]



The fork() System Call [3/3]

- Συνήθως οι διεργασίες γονέα και παιδιού επιτελούν διαφορετικούς ρόλους μετά την κλήση της fork():

```
pid_t pid;
pid = fork();

if (pid<0) {perror("Fork error"); exit(1);}

if (pid==0)  /* child */
{
  printf("I am the child process\n");
  while (1) putchar('c');
}
else         /* parent */
{
  printf("I am the parent process\n");
  while(1) putchar('p');
}
```

The Fork of Death!

- Προσοχή με τη fork()!!
  - Πανεύκολο να δημιουργήσετε χάος!! ☺

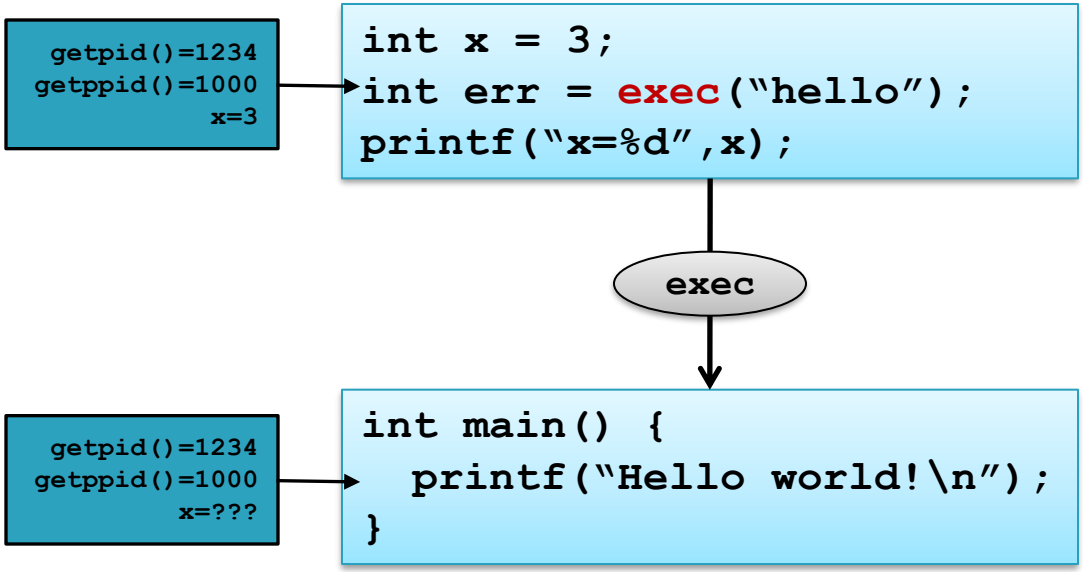
```
while (1) {
  fork();
}
```

- Ο αριθμός των διεργασιών ενός συστήματος Unix είναι πεπερασμένος
  - PID: 16-bit integer ➔ maximum 65536 processes!

- Μην χαίρεστε όμως! ☺
  - Οι administrators συνήθως λαμβάνουν τα μέτρα τους
  - Περιορίζουν το αριθμό processes ανά user
  - Πειραματιστείτε μόνο στο σπίτι ☺

The exec() System Call [1/4]

- Η **`exec()`** επιτρέπει σε μια διεργασία να αλλάξει το πρόγραμμα που εκτελεί (αλλά παραμένοντας **η ίδια** διεργασία!)
  - Code/Data for that process are destroyed
  - Οι environment variables διατηρούνται
  - Οι file descriptors διατηρούνται
  - Το νέο πρόγραμμα φορτώνεται και εκτελείται (από την αρχή)
  - Δεν υπάρχει επιστροφή!!



The exec() System Call [2/4]

- fork() and exec() μπορούν και να χρησιμοποιηθούν ξεχωριστά
  - Πότε;;;
- Συνήθως όμως, «πάνε πακέτο»!

```
if (fork()==0)     /* child */
{
  exec("hello");  /* load & execute new program */
  perror("Error calling exec()!\n");
  exit(1);
}
else  /* parent */
{
  ...
}
```

The exec() System Call [3/4]

- Τα command-line arguments περνιούνται σαν array of strings:
  - Το πρώτο, argv[0], πρέπει να είναι το program name
  - Το τελευταίο πρέπει να είναι NULL (σηματοδοτεί το τέλος του array)

- Αν δεν το τηρήσετε αυτό, **το πρόγραμμά σας μπορεί να έχει απρόβλεπτη συμπεριφορά**!
  - Μερικές φορές πολύ περίεργη!

- Π.χ., για να εκτελέσουμε **"ls -l /home/spyros"**,
  - Το όνομα προγράμματος είναι **"/bin/ls"**
  - Και το array of arguments είναι

    {"ls", "-l", "/home/spyros", NULL}

The exec() System Call [4/4]

- Υπάρχουν 4 versions του exec(), βάσει δύο χαρακτηριστικών:
  - Αναζήτηση του προγράμματος σε (i) absolute path, ή (ii) $PATH
  - Παράμετροι περασμένες ως **l**ist ή ως **v**ector (array)

	absolute path	$PATH
list (*n+1* parameters)	`execl()`	`execlp()`
vector (2 parameters)	`execv()`	`execvp()`

- Παράδειγμα execl():

```
execl("/bin/ls", "ls", "-l", "/home/spyros", NULL);
```

- Παράδειγμα execv():

```
char *params[4] = {"ls", "-l", "/home/spyros", NULL};
execv("/bin/ls", params);
```

Ερωτήσεις

- Q1: τι συμβαίνει όταν πληκτρολογείς μια εντολή στο shell;
  - Το shell process κάνει **fork()** (δηλ., δημιουργεί κλώνο του)
  - Το child process κάνει **exec()** για να εκτελέσει το ζητούμενο πρόγραμμα

- Q2: Κι αν η εντολή είναι σύνθετη, δηλαδή περιέχει pipes;;;

  π.χ.: `ls | grep ".txt" | wc`
  - Το shell κάνει fork() μία φορά για κάθε πρόγραμμα!
  - Δημιουργεί **pipes** και κανονίζει η έξοδος του ενός process να διοχετευθεί σαν είσοδος στο επόμενο, δημιουργώντας μια αλυσίδα διεργασιών

- Q3: Πως τερματίζει μια διεργασία;;
  - Δείτε τις επόμενες διαφάνειες! ☺

Τερματίζοντας μια διεργασία

- Μια διεργασία τερματίζεται (χωρίς σφάλμα) όταν:
  - Η `main()` function επιστρέφει

    - ή -
  - Το πρόγραμμα καλεί `exit()`

- Καμία διεργασία δεν μπορεί να τερματίσει («σκοτώσει») άλλη, ούτε καν ο ίδιος ο kernel!
  - Μπορεί μόνο να της στείλει ένα **signal**
    - …παρακαλώντας τη να τερματίσει!
  - Το signal θα καλέσει τον σχετικό **signal handler** (μια function)
  - By default, το signal **SIGINT** τερματίζει μια διεργασία
    - Αυτό είναι το σήμα που στέλνεται όταν πατάτε ^C
    - Όμως η διεργασία μπορεί να έχει αντικαταστήσει τον σχετικό handler
  - Το signal **SIGKILL** έχει το ίδιο αποτέλεσμα
    - Όμως δεν μπορεί να αντικατασταθεί ο handler του ☺

Inter-Process Communication

Inter-Process Communication

- Κανονικά, οι διεργασίες τρέχουν ανεξάρτητα, χωρίς να επηρεάζουν η μία την άλλη
  - Εκτελούνται σε απομόνωση (**executed in isolation**)
  - Ξεχωριστοί χώροι διευθύνσεων (**address spaces**)

- Υπάρχουν 4 μηχανισμοί που επιτρέπουν σε διεργασίες του **ίδιου υπολογιστή** να αλληλεπιδράσουν
  - **Signals:** Αποστολή ενός σήματος σε μια διεργασία (SIGINT, SIGKILL, κ.τ.λ.)
  - **Pipes:** Κανάλι επικοινωνίας για μεταφορά δεδομένων
  - **Shared memory:** Κοινή μνήμη στην οποία έχουν πρόσβαση πολλές διεργασίες
  - **Semaphores:** συγχρονισμός διεργασιών

- Αυτές οι μέθοδοι ονομάζονται **Inter-Process Communication** (IPC) και επιτρέπουν την επικοινωνία μεταξύ διεργασιών **του ίδιου υπολογιστή** (δηλ., **όχι** την επικοινωνία διεργασιών μέσω δικτύου)

Signal Handling [1/3]

- Μια διεργασία μπορεί να στείλει σε μια άλλη ένα signal
  - **SIGSEGV**: segmentation fault (non-authorized memory access)
  - **SIGBUS**: bus error (non-aligned memory access)
  - **SIGPIPE**: you tried to write in a pipe with no reader
  - **SIGCHLD**: one of your children processes has stopped
  - **SIGSTOP**, **SIGCONT**: pause and continue a process
  - **SIGUSR1**, **SIGUSR2**: two generic signals to be used by user programs
  - You can get a complete list of signals with **`kill -l`**

- Αυτό γίνεται με το system call kill()

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- **Αυτό δεν «σκοτώνει» (kill) απαραίτητα μια διεργασία!**

Signal Handling [2/3]

- Για έναν custom signal handler:

```
#include <signal.h>
void (*signal(int signum, void (*sighandler)(int)))(int);
```

informally: `HANDLER *signal(int signum, HANDLER *sighandler);`
where HANDLER would be a function like this: `void sighandler(int)`

- Π.χ.:

```
#include <signal.h>

void myhandler(int sig) {
  printf("I received signal number %d\n", sig);
  signal(sig, myhandler);  /* set it again it */
}

int main() {
  void (*oldhandler)(int);
  oldhandler = signal(SIGINT, myhandler);
  signal(SIGUSR1, myhandler);
  ...
}
```

Signal Handling [3/3]

- Όταν ένας signal handler τρέχει, το signal επαναφέρεται αυτόματα στην αρχική του συμπεριφορά (στον default signal handler)
  - Πρέπει να καλέσετε signal() **ξανά** μέσα στον signal handler!

- Προσοχή! Το Unix δεν κρατάει τα signals σε κάποια ουρά
  - Αν το ίδιο signal σταλεί σε μια διεργασία πολλές φορές σε σύντομα διαστήματα, **μπορεί και να αποσταλεί μόνο μία φορά!**

Zombie Processes

- Όταν μια διεργασία τερματίσει, δεν αφαιρείται κατευθείαν από το σύστημα
  - Η διεργασία γονέας μπορεί να ενδιαφέρεται για το return value
- Έτσι γίνεται «zombie»:
  - Η μνήμη κι οι πόροι της ελευθερώνονται μεν
  - Αλλά παραμένει η καταχώρησή της στο process table μέχρι ο γονέας να λάβει το return status
  - Αν ο γονέας έχει ήδη τερματίσει, η διεργασία «υιοθετείται» από την διεργασία init (με PID 1).
- Δεν πρέπει να αφήνετε zombies!!
  - Αλλιώς θα σας τελειώσουν τα PIDs
  - Ο administrator κανονικά πρέπει να θέτει περιορισμούς στο πόσες διεργασίες μπορεί να τρέχει ταυτόχρονα ένας χρήστης
- Για να τερματίσει ένα zombie, ο γονέας του πρέπει να καλέσει “wait”
  - Το wait() είναι blocking μέχρι να τερματίσει κάποιο απ’ τα παιδιά της διεργασίας
  - Εναλλακτικά, ο γονέας μπορεί να θέσει δικό του handler για το SIGCHLD signal, ώστε να ενημερωθεί όταν τερματίσει κάποιο παιδί του (και **τότε** να κάνει wait)

Waiting for Children Processes [1/2]

- Περιμένει (blocked) μέχρι κάποιο παιδί της διεργασίας να τερματίσει:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

- **`wait()`** περιμένει για οποιοδήποτε παιδί της διεργασίας
  - Εκτός αν κάποιο παιδί έχει ήδη τερματίσει (είναι zombie) οπότε επιστρέφει αμέσως
  - Επιστρέφει το PID του παιδιού που τερμάτισε
  - Το **`status`** περιέχει το return value του παιδιού

Waiting for Children Processes [2/2]

- Το waitpid() σου δίνει περισσότερο έλεγχο:

```
pid_t waitpid(pid_t pid, int *status, int option);
```

  - Το pid δηλώνει για **ποιο** παιδί να περιμένουμε (το -1 σημαίνει οποιοδήποτε)
  - Το option WNOHANG το κάνει non-blocking, ακόμα κι αν δεν έχει ήδη τερματίσει κανένα παιδί

- Παράδειγμα SIGCHLD signal handler

```
void sig_chld(int sig) {
  pid_t pid;
  int stat;
  while ( (pid=waitpid(-1, &stat, WNOHANG)) > 0 ) {
    printf("Child %d exited with status %d\n”, pid, stat);
  }
  signal(sig, sig_chld);
}
```

- **Ερώτηση:** Γιατί χρησιμοποιούμε while;

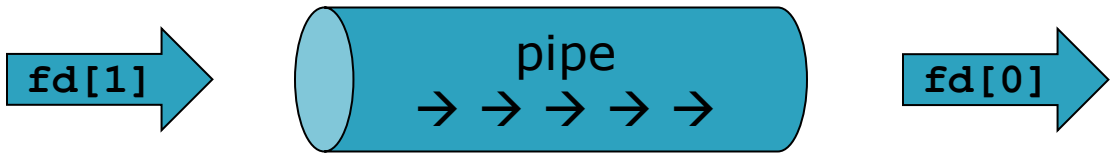
Pipes [1/3]

- Ένα pipe είναι ένα unidirectional communication channel ανάμεσα σε διεργασίες
  - Ό,τι γράψεις στο ένα άκρο, το διαβάζεις στο άλλο
  - Αν χρειάζεστε bidirectional επικοινωνία;
    - Χρησιμοποιήστε δύο pipes!

- Δημιουργώντας ένα pipe

```
#include <unistd.h>
int pipe(int fd[2]);
```

- Τα return parameters είναι:
  - **`fd[0]`** file descriptor για ανάγνωση
  - **`fd[1]`** file descriptor για γράψιμο



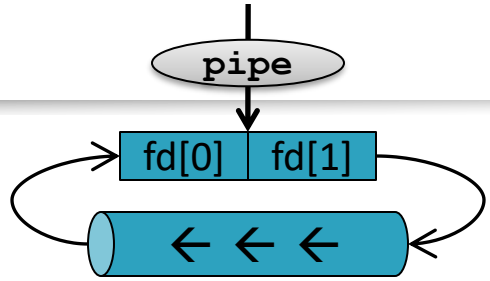
Pipes [2/3]

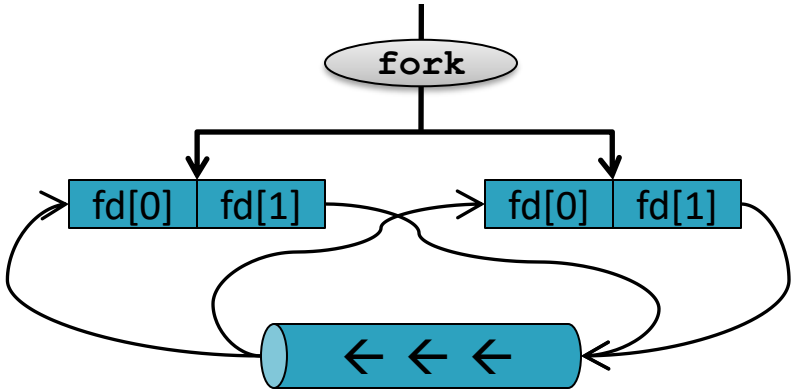
- Συνήθως τα pipes χρησιμοποιούνται σε συνδυασμό με fork()

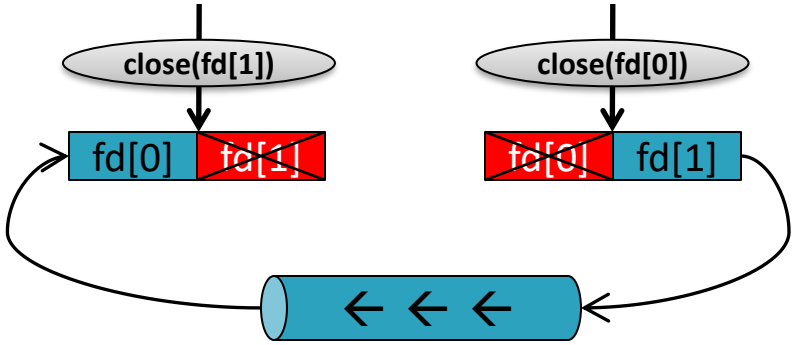
```
int main() {
  int pid, fd[2];
  char buf[64];

  if (pipe(fd)<0) exit(1);

  pid = fork();
  if (pid==0)          /* child */
  {
    close(fd[0]);    /* close reader */
    write(fd[1],"hello, world!",14);
  }
  else {               /* parent */
  {
    close(fd[1]);    /* close writer */
    if (read(fd[0],buf,64) > 0)
      printf("Received: %s\n", buf);
    waitpid(pid,NULL,0);
  }
}
```







Pipes [3/3]

- Για παράδειγμα, τα pipes χρησιμοποιούνται σε shell commands:

```
sort foo | uniq | wc
```

- Τα pipes επιτρέπουν μόνο την επικοινωνία ανάμεσα σε processes με κοινό «πρόγονο»
  - Διότι ένα pipe descriptor περνάει από τη μια διεργασία στην άλλη μόνο μέσα fork(), δηλαδή από γονέα σε παιδί

- Κι αν δυο διεργασίες που δεν έχουν κοινό πρόγονο θέλουν να εγκαταστήσουν pipe για επικοινωνία;
  - Υπάρχουν τα named pipes (γνωστά και ως FIFOs)
  - Είναι ειδικά αρχεία που συμπεριφέρονται ως pipes
  - Αρκεί οι δύο διεργασίες να συμφωνήσουν στο όνομα του αρχείου
    - Το mkfifo() δημιουργεί ένα named pipe
    - Μετά χρησιμοποιούμε open(), read(), write(), close()

Shared Memory [1/5]

- Κάνοντας χρήση **shared memory**, μπορούν πολλές διεργασίες να έχουν πρόσβαση σε μια **κοινή περιοχή μνήμης**

- Περιοχές shared memory πρέπει να
  - δημιουργηθούν (**created**)
  - γίνουν attached από κάποια διεργασία
  - γίνουν detached από την διεργασία
  - καταστραφούν (**destroyed**) όταν δεν χρειάζονται πλέον σε καμιά διεργασία

- Όταν χρησιμοποιούμε shared memory πρέπει να λαμβάνουμε τα μέτρα μας να αποφύγουμε **race conditions**, μέσω σημαφόρων (**semaphores**)

Shared Memory [2/5]

- Για να δημιουργήσουμε (**create**) ένα shared memory segment:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

  - **key**: rendezvous point (key=IPC_PRIVATE if it will be used by children processes)
  - **size**: size of the segment in bytes
  - **shmflg**: options (access control mask)
  - **Return value**: a shm identifier (or -1 for error)

Shared Memory [3/5]

- Για να κάνουμε **attach** ένα shared memory segment:

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

  - **shmid**: shared memory identifier (returned by shmget)
  - **shmaddr**: address where to attach the segment, or NULL if you don't care
  - **shmflg**: options (access control mask)
    - SHM_RDONLY: read-only

- Το **detach** a shared memory segment:

```
int shmdt(const void *shmaddr);
```

  - shmaddr: segment address
  - **Προσοχή**: το **shmdt()** δεν καταστρέφει την περιοχή shared memory!
  - Τα shared memory segments παραμένουν άθικτα ακόμα κι όταν όλες οι διεργασίες που τα χρησιμοποιούσαν έχουν τερματίσει!
    - Οφείλετε να τα καταστρέψετε όταν δεν χρειάζονται πλέον

Shared Memory [4/5]

- Για να καταστρέψουμε (**destroy**) ένα shared memory segment:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

  - **shmid**: shared memory identifier (returned by shmget)
  - **cmd=IPC_RMID** to destroy the segment
  - **buf**: NULL as far as we are concerned

- Command line
  - Με την εντολή **`ipcs`** μπορείτε να δείτε τα υπάρχοντα segments (και τα semaphores)
  - Μπορείτε και να τα καταστρέψετε με την εντολή

    `ipcrm shm <id>`

Shared Memory [5/5]

- Παράδειγμα:

```
int main() {
  int shmid = shmget(IPC_PRIVATE, sizeof(int), 0600);
  int *shared_int = (int *) shmat(shmid, 0, 0);
  *shared_int = 42;

  if (fork()==0) {
    printf("The value is: %d\n", *shared_int);
    *shared_int = 12;
    shmdt((void *) shared_int);
  }
  else {
    sleep(1);
    printf("The value is: %d\n", *shared_int);
    shmdt((void *) shared_int);
    shmctl(shmid, IPC_RMID, 0);
  }
}
```

Race Conditions [1/2]

- Όταν πολλές διεργασίες προσπελαύνουν κοινή μνήμη, πρέπει πάντα να θωρακίζετε το πρόγραμμά σας από race conditions!

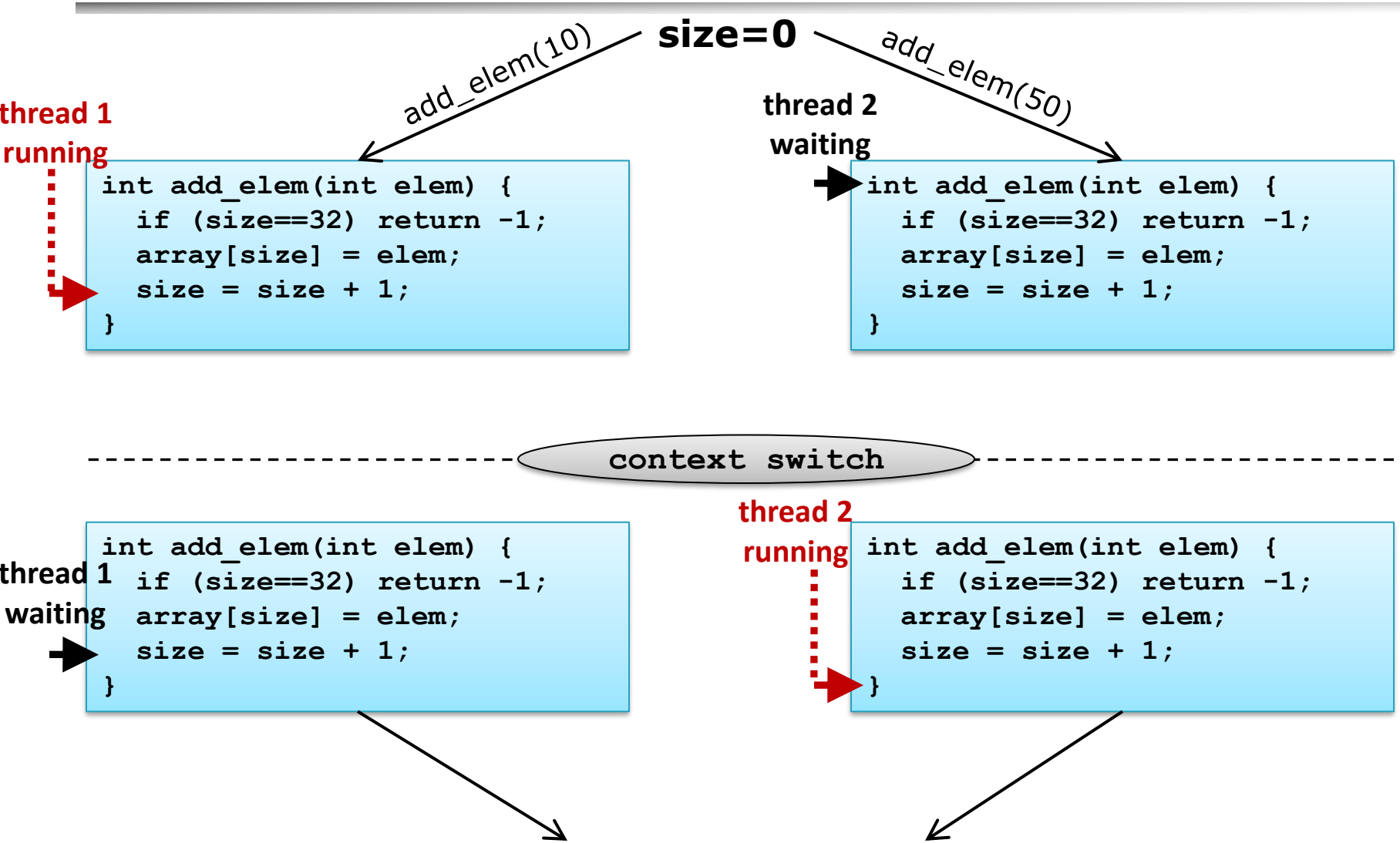
- Παράδειγμα: Προσθήκη στοιχείου σε array

```
int array[32], size;

int add_elem(int elem)
{
  if (size==32) return -1;
  array[size] = elem;
  size = size + 1;
}
```

- Ο παραπάνω κώδικας μπορεί να οδηγήσει σε **race condition**!
  - Αν δύο processes προσπαθήσουν να προσθέσουν στοιχείο ταυτόχρονα, μπορεί να προκληθεί σφάλμα

Race Conditions [2/2]



Semaphores [1/6]

- Σε πολλές περιπτώσεις χρειαζόμαστε να συγχρονίσουμε την εκτέλεση δύο ή περισσότερων διεργασιών όταν:
  - μοιράζονται κοινούς πόρους (κοινή μνήμη, file descriptors, devices, κτλ.)
  - μια διεργασία πρέπει να περιμένει κάποιο event (από κάποια άλλη)

- Μία σημαφόρος (**semaphore**) αποτελείται από
  - έναν **θετικό ακέραιο** και
  - δύο μεθόδους: **UP()** και **DOWN()**
    - **DOWN():**
      - if (sem>=1) then {sem=sem-1}
      - else {block the process}
    - **UP():**
      - if (there are blocked processes) then {unblock one of them}
      - else {sem=sem+1}

- Οι μέθοδοι up() και down() **εκτελούνται ατομικά**!
  - Δηλαδή, δεν μπορούν να «διακοπούν» στη μέση

Semaphores [2/6]

- Οι σημαφόροι χρησιμεύουν σε δύο σενάρια:
  - **Για αμοιβαίο αποκλεισμό** (**mutual exclusion**): όταν δεν πρέπει να εκτελέσουν κάποιο κρίσιμο κομμάτι κώδικα (**critical region**) περισσότερες από μία διεργασίες ταυτόχρονα
    - Ξεκινάμε με **sem=1**
    - Μπαίνοντας στο critical region: **DOWN(sem)**
    - Βγαίνοντας από το critical region: **UP(sem)**
  - **Για συγχρονισμό διεργασιών** (**process synchronization**): όταν μια διεργασία πρέπει να περιμένει ένα event άλλης διεργασίας
    - Ξεκινάμε με **sem=0**
    - Για να περιμένουμε για το event: **DOWN(sem)**
    - Για να κάνουμε **trigger** το event: **UP(sem)**
    - Τι συμβαίνει αν το event γίνει triggered πριν η άλλη διεργασία κάνει DOWN();
- **Ερώτηση:** Πως μπορούμε να αποφύγουμε το race condition στην add_elem();
- **Ερώτηση:** Πως μπορούμε να επιτρέψουμε μέχρι και k>1 διεργασίες ταυτόχρονα σε ένα critical region;

Semaphores [3/6]

- Οι σημαφόροι δημιουργούνται σε arrays
- Για να δημιουργήσουμε (**create**) ένα array of semaphores:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

  - **key**: rendezvous point or IPC_PRIVATE
  - **nsems**: number of semaphores to create
  - **semflg:** access rights
  - **Return value:** a semaphore array identifier

Semaphores [4/6]

- Τα UP() και DOWN() γίνονται μέσω της ίδιας function:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

  - **semid**: the semaphore array identifier
  - **sops**: an array of commands to be issued
  - **nsops**: the size of sops

- To sembuf struct:

```
struct sembuf {
  short sem_num;  /* semaphore number: 0 = first */
  short sem_op;   /* semaphore operation */
  short sem_flg;  /* operation flags */
};
```

  - **sem_op**: 1 for UP(), -1 for DOWN()
  - **Όλες οι λειτουργίες εκτελούνται ατομικά, σαν μία.**

Semaphores [5/6]

- Διαχειριζόμαστε τις σημαφόρους ως εξής:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

  - **semid**: the semaphore array identifier
  - **semnum**: the semaphore number
  - **cmd**: command
    - IPC_RMID: **destroy** the semaphore
    - GETVAL: return the **value** of the semaphore

- Οι σημαφόροι «ζουν» και μετά το πέρας των διεργασιών που τις δημιούργησαν και/ή τις χρησιμοποίησαν
  - Πρέπει να απελευθερώσετε (destroy) τις σημαφόρους όταν δεν χρειάζονται πλέον
  - Είτε μέσα απο το πρόγραμμά σας, ή από το command line με ipcs, ipcrm

Semaphores [6/6]

- Παράδειγμα:

```
int main() {
  struct sembuf up   = {0,  1, 0};
  struct sembuf down = {0, -1, 0};
  int my_sem, v1, v2, v3, v4;

  my_sem = semget(IPC_PRIVATE, 1, 0600); /* create semaphore */
  v1 = semctl(my_sem, 0, GETVAL);

  semop(my_sem, &up, 1);                 /* UP() */
  v2 = semctl(my_sem, 0, GETVAL);

  semop(my_sem, &down, 1);               /* DOWN() */
  v3 = semctl(my_sem, 0, GETVAL);

  semctl(my_sem, 0, IPC_RMID);           /* destroy */
  v4 = semctl(my_sem, 0, GETVAL);

  printf("Semaphore values: %d %d %d %d\n", v1, v2, v3, v4);
}
```

Posix Threads

Threads vs. Processes

- Τα multi-process προγράμματα είναι δαπανηρά:
  - Το fork() χρειάζεται να αντιγράψει τη μνήμη της διεργασίας
  - Το interprocess communication είναι δύσκολο

- **Threads** (**Νήματα**): “lightweight processes”
  - Κάθε process περιέχει ένα ή περισσότερα threads
  - Όλα τα threads ενός process εκτελούν το ίδιο πρόγραμμα
  - Όλα τα threads μοιράζονται τις ίδιες machine-code instructions, την ίδια global memory, τα ίδια open files, και τους ίδιους signal handlers
  - Κάθε thread έχει δικό του thread ID, program counter (PC), stack και stack pointer (SP), errno, και signal mask

Threads in C and Java

- Posix Threads
  - Posix Threads (pthreads) are standard among Unix systems
    - Also available on Windows through 3rd party libraries (Pthreads-w32)
  - The operating system must have special support for threads
    - Linux, Solaris, and virtually all Unix systems have it
  - Programs must be linked with –lpthread
    - Beware: Solaris will compile fine even if you forget the –lpthread (but your program will not work)

- Java Threads
  - Threads are a native feature of Java: every virtual machine has thread support
  - They are portable on any Java platform
  - Java threads can be:
    - mapped to operating system threads (kernel threads or native threads)
    - or emulated in user space (user threads or green threads)

Δημιουργία pthread

- Για να δημιουργήσουμε ένα νέο pthread:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

  - **thread**: thread id (this is a return argument)
  - **attr**: attributes (i.e., options)
  - **start_routine**: function that the thread will execute
  - **arg**: parameter to be passed to the thread
  - Return value: 0 on success, error value on failure

- To initialize and destroy the default pthread attributes

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

Τερματισμός ενός pthread

- Ένα pthread τερματίζει όταν:
  - Τερματίζει το process του
  - Τερματίζει το thread-γονέας του
  - Τερματίζει η `start_routine()` του
  - Καλεί ρητά το pthread_exit():

```
#include <pthread.h>
void pthread_exit(void *retval);
```

- Like processes, stopped threads must be waited for:

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

pthread Create/Delete Example

- To create a pthread:

```
#include <pthread.h>

void *func(void *param) {
  int *p = (int *) param;
  printf("New thread: param=%d\n",*p);
  return NULL;
}

int main() {
  pthread_t id;
  pthread_attr_t attr;
  int x = 42;

  pthread_attr_init(&attr);
  pthread_create(&id, &attr, func, (void *) &x);
  pthread_join(id, NULL);
}
```

Detached Threads

- A “detached” thread:
  - does not need to be joined by `pthread_join()`
  - does not stop when its parent thread stops
- By default, threads are “joinable” (i.e. “attached”)
- To create a detached thread, set an attribute before creating the thread:

```
pthread_t id;
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&id, &attr, func, NULL);
```

- You can also detach a thread later with `pthread_detach()`
  - But you cannot reattach it!

Race Conditions with Threads

- Τα threads μοιράζονται τα περισσότερα resources
  - memory , file descriptors, κτλ.

- **Προσοχή!! Κίνδυνος - Θάνατος!!**
  - Είναι πολύ εύκολο να δημιουργήσετε race conditions με threads, χωρίς καν να το καταλάβετε

- Πρέπει πάντα να ερευνάτε αν χρειάζεται συγχρονισμός
  - And solve them with special thread synchronization primitives

- Pthreads have two synchronization concepts:
  - **Mutex**
  - **Condition Variables**

Pthread Sync with Mutex [1/2]

- Mutex (mutual exclusion)

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Pthread Sync with Mutex [2/2]

- Example

```
pthread_mutex_t mutex;

int add_elem(int elem) {
  int n;
  pthread_mutex_lock(&mutex);
  if (size==32) {
    pthread_mutex_unlock(&mutex);
    return -1;
  }
  array[size++] = elem;
  n = size;
  pthread_mutex_unlock(&mutex);
}

int main() {
  pthread_mutexattr_t attr;
  pthread_mutexattr_init(&attr);
  pthread_mutex_init(&mutex, &attr);
  ...
  pthread_mutex_destroy(&mutex);
}
```

Thread Safety with Unix Primitives

- Προσοχή!
  - Κάποια Unix primitives και library functions έχουν σχεδιαστεί **χωρίς** να υποστηρίζουν **thread safety**!
  - Όταν γράφετε ένα multi-threaded πρόγραμμα, οφείλετε να ελέγχετε τα man pages **αν οι συναρτήσεις που καλείτε είναι thread-safe**

- Π.χ.:
  - **`gethostbyname()`**: δεν είναι thread-safe!
  - Μπορείτε να τη χρησιμοποιήσετε σε multi-threaded προγράμματα
    - …αλλά με **mutex**!

Condition Variables

- **Condition Variables**
  - Ο μηχανισμός που επιτρέπει σε ένα thread να περιμένει (blocked) μέχρι να το «ξυπνήσει» κάποιο άλλο thread

- Τα condition variables χρειάζονται mutex για να λειτουργήσουν
  - The mutex is used to synchronize access to the condition variable
  - Όταν «ξυπνάνε» πρέπει να ελέγχουν αν πρέπει όντως να ξυπνήσουν, ή αν πρέπει να συνεχίσουν να περιμένουν.

- Για περισσότερες πληροφορίες:

  http://www.ibm.com/developerworks/linux/library/l-posix3/

  https://computing.llnl.gov/tutorials/pthreads/

API for Condition Variables

- At init time:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;
int predicate = FALSE;

pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond_var, NULL);
```

- To wait for an event:

```
pthread_mutex_lock(&mutex);
while (predicate==FALSE)
  pthread_cond_wait(&cond_var,&mutex);
predicate = FALSE;
pthread_mutex_unlock(&mutex);
```

- To trigger an event:

```
pthread_mutex_lock(&mutex);
predicate = TRUE;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```