Προγραμματισμός ΙΙ (ECE116)

#19

σήματα (signals)

Σπύρος Λάλης
lalis@uth.gr

Τι είναι ένα σήμα;

- Το σήμα είναι μια **ειδοποίηση** που στέλνεται στην διεργασία που εκτελεί ένα πρόγραμμα
- Για πολλούς και διαφορετικούς λόγους
  - πρόσβαση σε άκυρη θέση μνήμης
  - πρόβλημα σε αριθμητικής πράξη
  - τερματισμός διεργασίας παιδιού
  - γράψιμο σε αγωγό από τον οποίο δεν πρόκειται να διαβάσει κανείς
  - αποστολή σήματος από άλλη διεργασία
- Κάθε σήμα έχει ως αναγνωριστικό έναν **αριθμό**
  - φανερώνει τον **τύπο** τους / **λόγο** για τον οποίο δημιουργήθηκαν
- Για κάθε σήμα υπάρχει **προκαθορισμένος αυτόματος** χειρισμός από το λειτουργικό σύστημα
  - και δυνατότητα χειρισμού από το πρόγραμμα!

Σήματα σφάλματος

- `SIGFPE`: εσφαλμένη αριθμητική λειτουργία (T+)
- `SIGILL`: άκυρη εντολή (T+)
- `SIGSYS`: άκυρη κλήση συστήματος (T+)
- `SIGBUS`: άκυρη διεύθυνση σε επίπεδο υλικού (T+)
- `SIGSEGV`: άκυρη αναφορά μνήμης (T+)
- `SIGPIPE`: γράψιμο σε αγωγό χωρίς αναγνώστη (T)

Αυτόματες/προεπιλεγμένες ενέργειες (αν δεν γίνει χειρισμός από το πρόγραμμα):
Π: παράβλεψη
T: τερματισμός
T+: τερματισμός + άλλες ενέργειες, π.χ., δημιουργία core dump
Α: αναστολή/σταμάτημα εκτέλεσης
Σ: συνέχιση εκτέλεσης (μετά από αναστολή)

Σήματα διακοπής/τερματισμού κλπ

- `SIGINT`: διακοπή από το πληκτρολόγιο (Τ)
- `SIGQUIT`: εγκατάλειψη από το πληκτρολόγιο (Τ+)
- `SIGHUP`: κλείσιμο γραμμής/τερματικού (Τ)
- `SIGTERM`: τερματισμός (Τ)
- `SIGKILL`: θανάτωση (Τ)

Αυτόματες/προεπιλεγμένες ενέργειες (αν δεν γίνει χειρισμός από το πρόγραμμα):
Π: παράβλεψη
Τ: τερματισμός
Τ+: τερματισμός + άλλες ενέργειες, π.χ., δημιουργία core dump
Α: αναστολή εκτέλεσης
Σ: συνέχιση εκτέλεσης (μετά από αναστολή)

Σήματα ελέγχου κατάστασης διεργασίας

- `SIGCHLD`: αλλαγή κατάστασης παιδιού (Π)
- `SIGSTOP`: αναστολή εκτέλεσης διεργασίας (Α)
- `SIGCONT`: συνέχιση εκτέλεσης διεργασίας (Σ)
- `SIGTTIN`: ανάγνωση από το παρασκήνιο (Α)
- `SIGTTOU`: γράψιμο από το παρασκήνιο (Α)

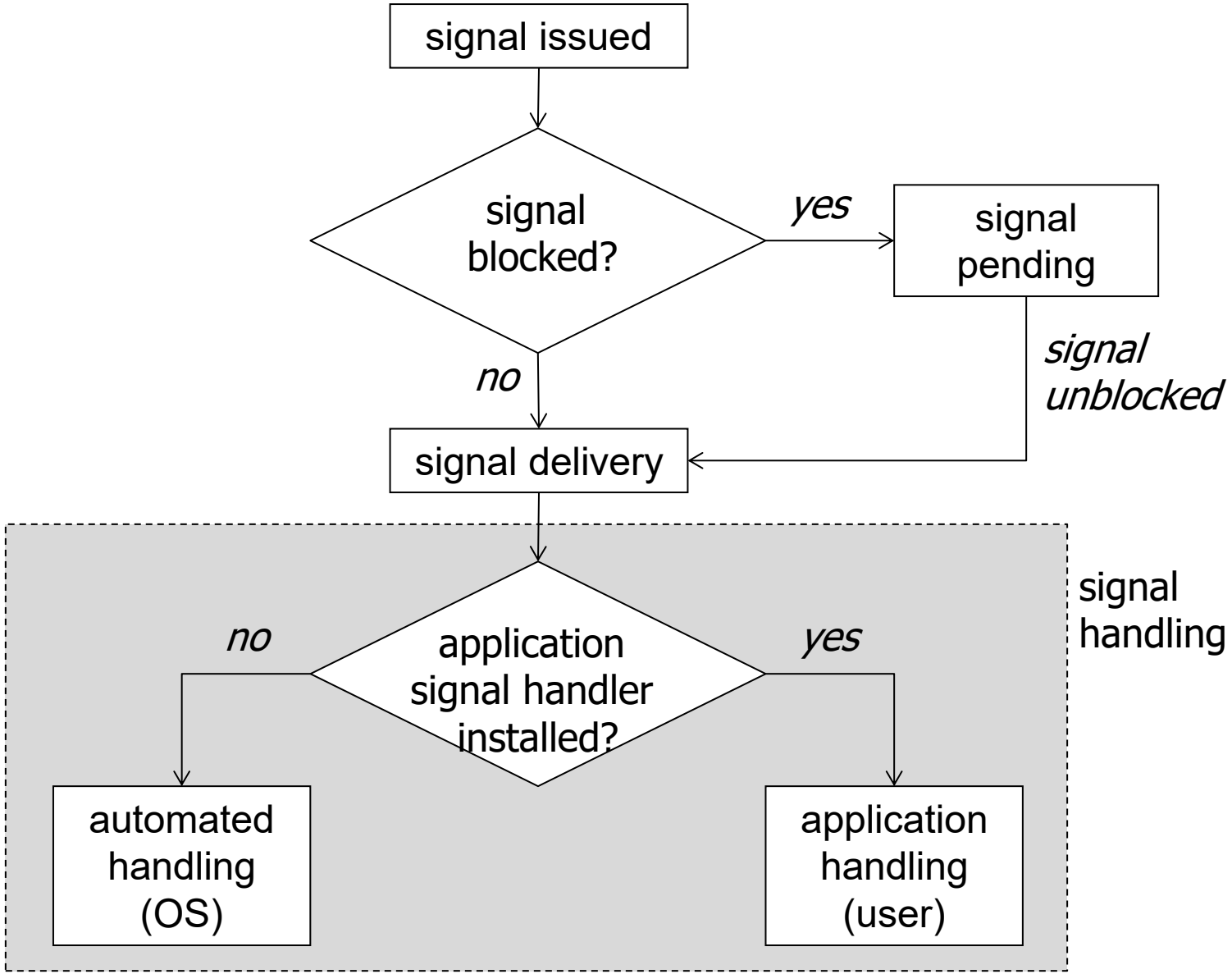
Αυτόματες/προεπιλεγμένες ενέργειες (αν δεν γίνει χειρισμός από το πρόγραμμα):
Π: παράβλεψη
Τ: τερματισμός
Τ+: τερματισμός + άλλες ενέργειες, π.χ., δημιουργία core dump
Α: αναστολή εκτέλεσης
Σ: συνέχιση εκτέλεσης (μετά από αναστολή)

Σύγχρονα και ασύγχρονα σήματα

- Ο χαρακτηρισμός ενός σήματος ως **σύγχρονου** ή **ασύγχρονου**, φανερώνει το αν το σήμα μπορεί να προκληθεί από μια **ρητή** ενέργεια της ίδιας της διεργασίας μέσω του κώδικα που αυτή εκτελεί
- **Σύγχρονα** σήματα: προκύπτουν ως αποτέλεσμα μιας ενέργειας που επιχείρησε **η ίδια η διεργασία**
  - π.χ., εξαίρεση αριθμητικής πράξης (`SIGFPE`), άκυρη θέση μνήμης (`SIGSEGV`), άκυρη εντολή μηχανής (`SIGILL`), γράψιμο σε αγωγό που δεν έχει αναγνώστες (`SIGPIPE`) ...
- **Ασύγχρονα** σήματα: προκύπτουν **χωρίς** να τα έχει προκαλέσει η ίδια η διεργασία με άμεσο τρόπο
  - π.χ., διακοπή εκτέλεσης (`SIGINT`), αναστολή διεργασίας (`SIGSTOP`), συνέχιση εκτέλεσης (`SIGCONT`), τερματισμός ή αλλαγή κατάστασης διεργασίας παιδιού (`SIGCHLD`) ...

Κύκλος ζωής ενός σήματος

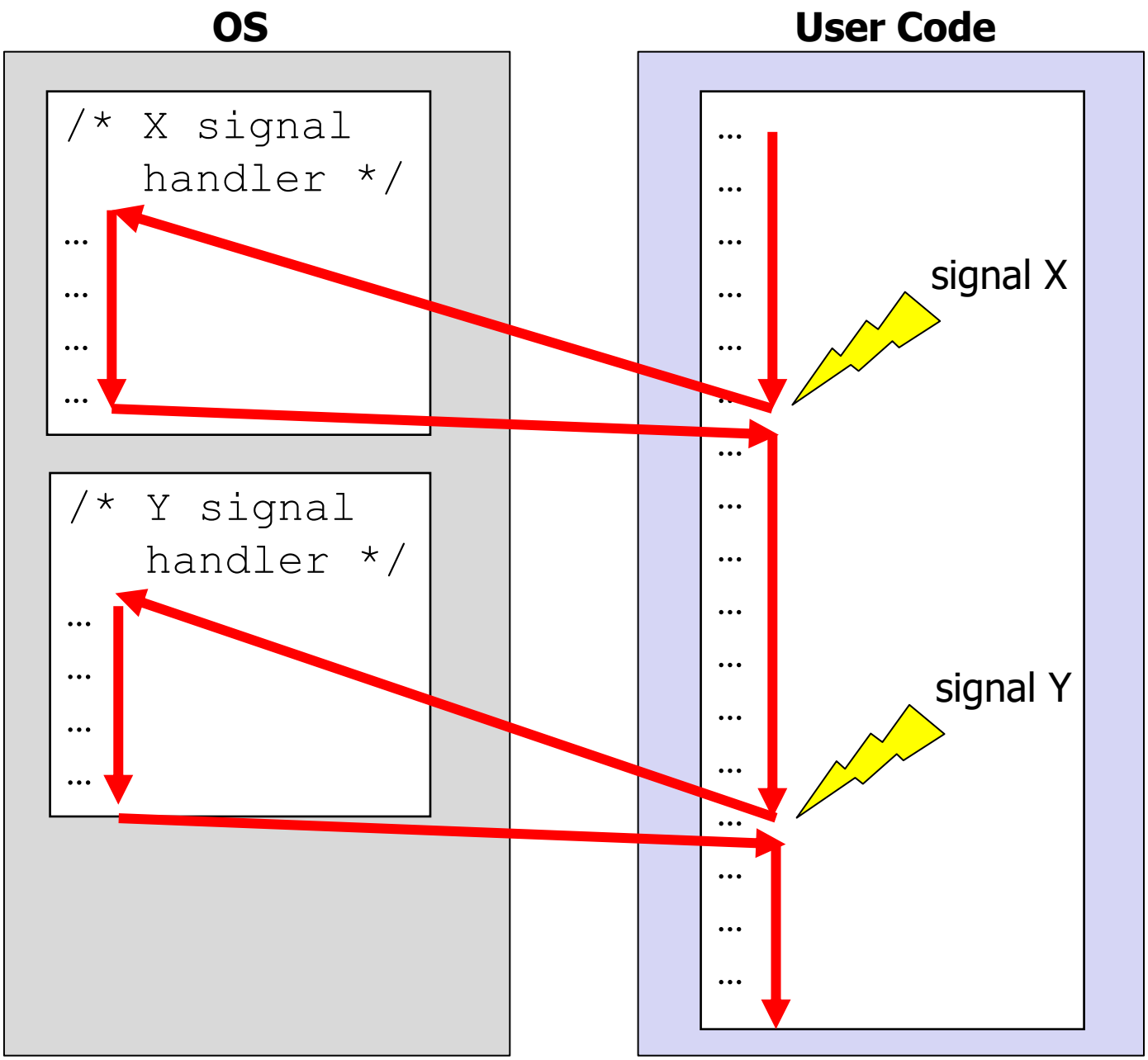
- **Γέννηση:** δημιουργία/αποστολή σήματος
- **Μπλοκάρισμα:** (εφόσον έχει ενεργοποιηθεί) αναστολή παράδοσης/χειρισμού του σήματος
- **Παράδοση/παραλαβή:** χρονική στιγμή που γίνεται επεξεργασία του σήματος στο πλαίσιο της διεργασίας
- **Χειρισμός:** διαδικασία επεξεργασίας του σήματος


signal issued
signal blocked?
yes
signal pending
signal unblocked
no
signal delivery
signal handling
no
application signal handler installed?
yes
automated handling (OS)
application handling (user)

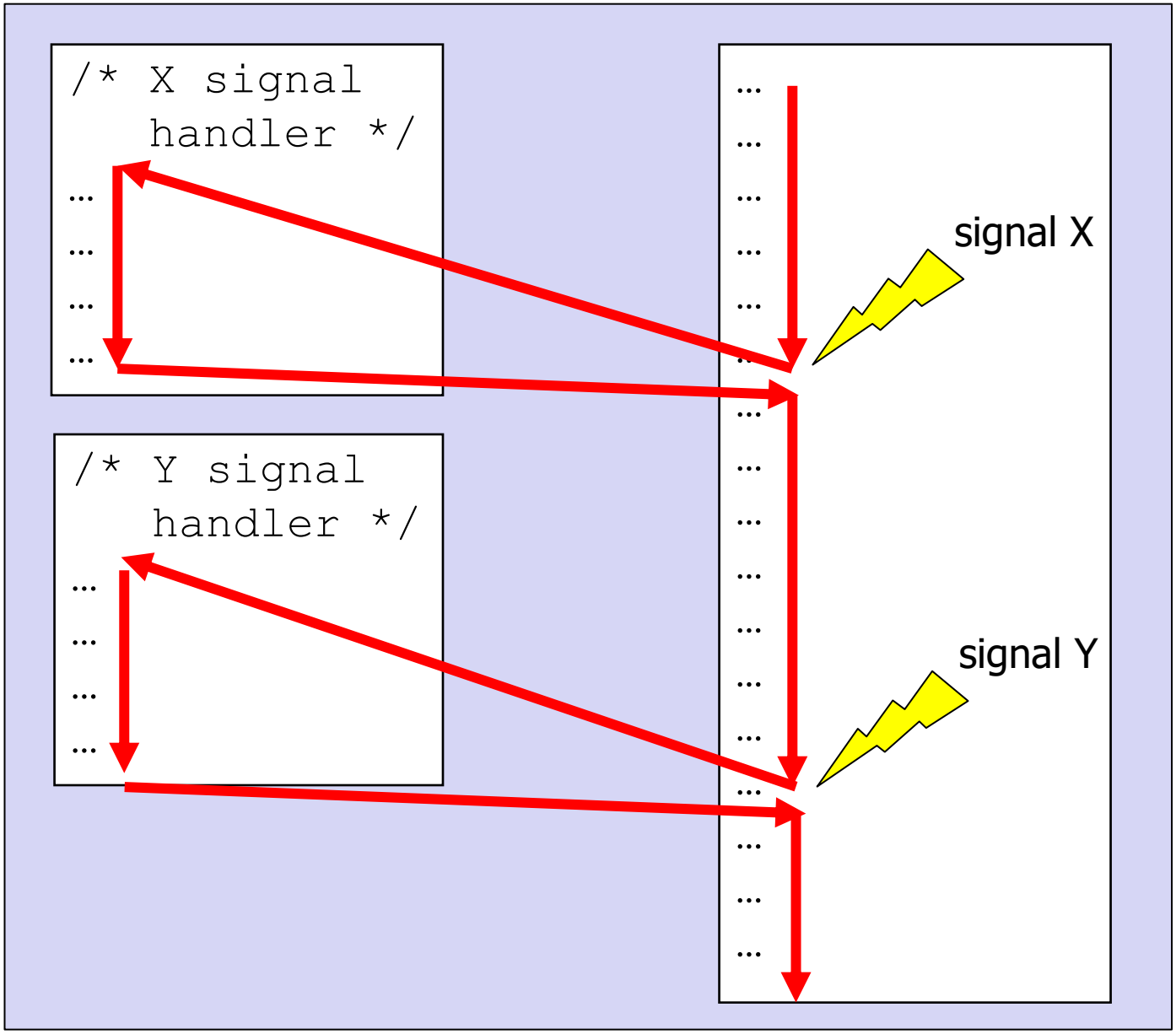
Χειρισμός σήματος

- Το ΛΣ ορίζει **αυτόματες προκαθορισμένες** ενέργειες χειρισμού για κάθε τύπο σήματος
  - **παράβλεψη** σήματος (π.χ., για `SIGCHLD`)
  - **αναστολή** εκτέλεσης διεργασίας (π.χ., για `SIGSTOP`)
  - **συνέχεια** εκτέλεσης διεργασίας (π.χ., για `SIGCONT`)
  - **τερματισμός** διεργασίας (π.χ., για `SIGINT`)

- Μια διεργασία μπορεί να **«παγιδέψει»** ένα σήμα, και να **αλλάξει** τον χειρισμό του
  - αυτόματος προκαθορισμένος χειρισμός σήματος (`SIG_DFL`)
  - αυτόματη παράβλεψη σήματος (`SIG_IGN`)
  - ρητός χειρισμός σήματος από **κώδικα** του προγράμματος (**χειριστής σήματος – signal handler**)

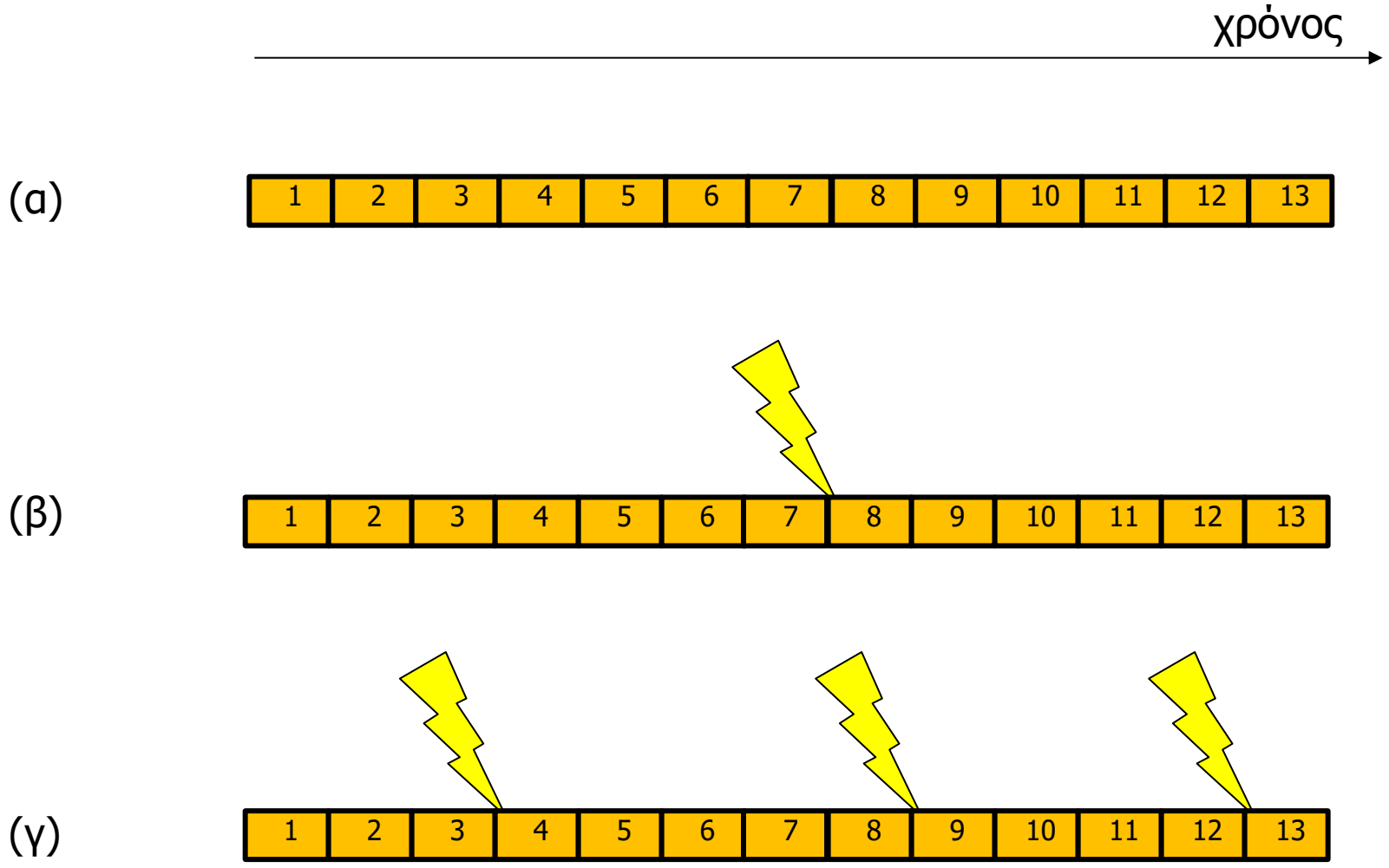

OS
User Code
/* X signal handler */
/* Y signal handler */
signal X
signal Y

User Code



Διακοπή ροής εκτέλεσης διεργασίας

- Ο χειρισμός σημάτων **διακόπτει** την κανονική ροή εκτέλεσης της διεργασίας
- **Σύγχρονα σήματα:** η διακοπή/χειρισμός γίνεται **στο σημείο** που εκτελέστηκε η ενέργεια που προκάλεσε το σήμα
- **Ασύγχρονα σήματα:** η διακοπή/χειρισμός μπορεί να προκύψει σε **οποιοδήποτε** σημείο του προγράμματος
- **Δεν** γνωρίζουμε **πόσες φορές** μια διεργασία θα λάβει ένα ασύγχρονο σήμα, **ούτε το σημείο** της εκτέλεσης όπου θα βρίσκεται όταν λάβει το σήμα
- **Δεν** εξαρτάται (καθόλου) από τον κώδικα που εκτελεί η διεργασία


χρόνος
(α)
1 2 3 4 5 6 7 8 9 10 11 12 13
(β)
1 2 3 4 5 6 7 8 9 10 11 12 13
(γ)
1 2 3 4 5 6 7 8 9 10 11 12 13

Παραγωγή/αποστολή σήματος

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

- `pid > 0`: στέλνει το σήμα `signum` στην διεργασία με αναγνωριστικό `pid`
- `pid == 0`: στέλνει το σήμα `signum` στην ομάδα διεργασιών το αναγνωριστικό της οποίας είναι το ίδιο με το αναγνωριστικό ομάδας του αποστολέα
- `pid < -1`: στέλνει το σήμα `signum` στην ομάδα διεργασιών το αναγνωριστικό της οποίας είναι ίσο με την απόλυτη τιμή του `pid`
- `pid == -1`: στέλνει το σήμα `signum` σε όλες τις διεργασίες για τις οποίες έχει άδεια ο αποστολέας

```
int main(int argc, char *argv[]) {
  int pid;

  pid = fork();

  if (pid == 0) {
    while (1) {
      sleep(1);
      printf("child: ping\n");
    }
    return(0);
  }

  sleep(5);
  printf("parent: send SIGINT to child\n");
  kill(pid, SIGINT);
  return(0);
}
```

Καθορισμός ενεργειών χειρισμού σήματος

```
int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oact);
```

- Καθορίζει ενέργειες χειρισμού για το σήμα `signum`
- Οι **νέες** ενέργειες προσδιορίζονται στην δομή `act`
- Οι **παλιές** ενέργειες αποθηκεύονται στην δομή `oact`
  - για μπορεί να γίνει επαναφορά της προηγούμενης κατάστασης
- Μπορεί να καθοριστούν διαφορετικές ενέργειες για κάθε ξεχωριστό σήμα
- Οι ενέργειες για `SIGKILL` και `SIGSTOP` **δεν** αλλάζουν

Δομή ενεργειών χειρισμού σήματος

```
struct sigaction {
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_sigaction)(int,siginfo_t*,void *);
};
```

- Στο `sa_handler` ανατίθεται ο **χειριστής σήματος**
  - `SIG_IGN` ή `SIG_DFL` ή δείκτης σε συνάρτηση
- Το `sa_mask` καθορίζει τα σήματα που **μπλοκάρονται** όταν εκτελείται ο χειριστής σήματος
  - πλέον του σήματος που προκάλεσε τον χειρισμό του σήματος, και όσων άλλων σημάτων είναι ήδη μπλοκαρισμένα
- Το `sa_flags` καθορίζει επιπλέον ιδιότητες
  - π.χ. `SA_RESTART` για επανεκτέλεση κλήσεων συστήματος

```
int main(int argc, char *argv[]) {
  struct sigaction oldaction, action = {{0}};
  int i;

  action.sa_handler = SIG_IGN;

  sigaction(SIGINT, &action, &oldaction);
  printf("changed SIGINT handling to ignore\n");

  for (i=0; i<3; i++) {
    printf("going to sleep\n");
    sleep(10);
  }

  sigaction(SIGINT, &oldaction, NULL);
  printf("changed SIGINT handling back to default\n");

  printf("going to sleep\n");
  sleep(10);

  return(0);
}
```

ignore signal
(no user code involved)

```c
static void myhandler(int sig) {
  write(STDOUT_FILENO,"got SIGINT\n",11);
}

int main(int argc, char *argv[]) {
  struct sigaction oldaction, action = {{0}};
  int i;

  action.sa_handler = myhandler;

  sigaction(SIGINT, &action, &oldaction);
  printf("changed SIGINT handling to myhandler\n");

  for (i=0; i<3; i++) {
    printf("going to sleep\n");
    sleep(10);
  }

  sigaction(SIGINT, &oldaction, NULL);
  printf("changed SIGINT handling back to default\n");

  printf("going to sleep\n");
  sleep(10);

  return(0);
}
```

user-level signal handler

Αναμονή παραλαβής/χειρισμού σήματος

```
int sleep(int secs);
```

- Μπλοκάρει μέχρι να γίνει **ρητός χειρισμός** ενός σήματος **από την εφαρμογή** ή μετά την πάροδο του χρονικού διαστήματος `secs`
  - επιστρέφει 0 αν η διεργασία δεν διακόπηκε από σήμα
  - διαφορετικά, το χρονικό διάστημα που απέμεινε

```
int pause();
```

- Μπλοκάρει μέχρι να γίνει **ρητός χειρισμός** ενός σήματος **από την εφαρμογή**
  - επιστρέφει `-1` με την `errno` ίση με `EINTR`
- Οι `pause` και `sleep` **δεν** επιστρέφουν (πρόωρα) όταν γίνεται αυτόματος χειρισμός σήματος (χωρίς να εκτελεστεί χειριστής σήματος της εφαρμογής)

Διακοπή κλήσεων συστήματος

- Η παράδοση και χειρισμός ενός ασύγχρονου σήματος μπορεί να **διακόψει** κάποιες **κλήσεις συστήματος**
- Συνήθως, αυτές που **μπλοκάρουν** την διεργασία

- Εφόσον γίνει ρητός χειρισμός του σήματος από το πρόγραμμα, η κλήση συστήματος που διακόπηκε **αποτυγχάνει**
  - η `errno` παίρνει την τιμή `EINTR`

- Αν αυτό δεν είναι επιθυμητό, χρησιμοποιούμε την επιλογή `SA_RESTART` κατά την εγκατάσταση του χειριστή σήματος ή την `siginterrupt`

```
static void myhandler(int sig) {
  write(STDOUT_FILENO,"got SIGINT\n",11);
}

int main(int argc, char *argv[]) {
  int n; char c;
  struct sigaction action = {{0}};

  action.sa_handler = myhandler;
  sigaction(SIGINT,&action,NULL);

  while (1) {
    n = read(STDIN_FILENO,&c,1);
    if (n > 0) { write(STDOUT_FILENO,&c,1); }
    else if (n == 0) { printf("end of input\n"); break; }
    else if (errno != EINTR) { perror("read"); break; }
    else { printf("read interrupted, retrying\n"); }
  }
  return(0);
}
```

```
static void myhandler(int sig) {
  write(STDOUT_FILENO,"got SIGINT\n",11);
}

int main(int argc, char *argv[]) {
  int n; char c;
  struct sigaction action = {{0}};

  action.sa_handler = myhandler;
  action.sa_flags = SA_RESTART;
  sigaction(SIGINT,&action,NULL);

  while (1) {
    n = read(STDIN_FILENO,&c,1);
    if (n > 0) { write(STDOUT_FILENO,&c,1); }
    else if (n == 0) { printf("end of input\n"); break; }
    else if (errno != EINTR) { perror("read"); break; }
    else { printf("read interrupted, retrying\n"); }
  }
  return(0);
}
```

automatically restart interrupted system calls

Περισσότερα για την `waitpid`

- Όταν μια διεργασία παιδί αλλάζει την κατάσταση εκτέλεσης, στέλνεται στον γονιό το σήμα `SIGCHLD`
  - κατά σύμβαση αυτό απλά αγνοείται
- Ο γονιός μπορεί να **περιμένει** για μια τέτοια **αλλαγή κατάστασης** (και) μέσω της `waitpid`
  - δίνοντας τις επιλογές `WUNTRACED/WCONTINUED`
- Έλεγχος της κατάστασης του παιδιού μπορεί να γίνει, στην συνέχεια, μέσω αντίστοιχων μακροεντολών
- `WIFSTOPPED()`: `true` αν το παιδί ανέστειλε την εκτέλεση του (χωρίς να έχει τερματιστεί)
- `WSTOPSIG()`: το σήμα που οδήγησε στην αναστολή της εκτέλεσης του παιδιού
- `WIFCONTINUED()`: `true` αν το παιδί συνεχίζει την εκτέλεση του λόγω `SIGCONT`

```
static void myhandler(int sig) {
  write(STDOUT_FILENO,"got SIGCHLD\n",12);
}

int main(int argc, char *argv[]) {
  pid_t pid; int status, i;
  struct sigaction action = {{0}};

  action.sa_handler = myhandler;
  sigaction(SIGCHLD, &action, NULL);

  if (!(pid=fork())) {
    for (i=0; i<20; i++) { sleep(3); printf("ping\n"); }
    return(42);
  }

  while (1) {
    waitpid(pid,&status,WUNTRACED|WCONTINUED);
    if (WIFEXITED(status)) {
      printf("child returned %d\n", WEXITSTATUS(status));
      break;
    }
    else if (WIFSIGNALED(status)) {
      printf("child terminated %d\n", WTERMSIG(status));
      break;
    }
    else if (WIFSTOPPED(status)) {
      printf("child stopped %d\n", WSTOPSIG(status));
    }
    else if (WIFCONTINUED(status)) {
      printf("child continued\n");
    }
  }
  return(0);
}
```

Μπλοκάρισμα σημάτων κατά την κανονική ροή εκτέλεσης του προγράμματος

- Σε κάποια σημεία του προγράμματος, μπορεί να μην είναι επιθυμητή η παράδοση και ο χειρισμός σημάτων
- Τα σήματα μπορεί να **μπλοκαριστούν**
  - μέσω της **μάσκας μπλοκαρίσματος** της διεργασίας
- Τα μπλοκαρισμένα σήματα **δεν παραδίδονται** στην διεργασία για χειρισμό
- Παραμένουν σε **εκκρεμότητα** μέχρι να ξεμπλοκαριστούν
- Ο χειρισμός ενός σήματος που βρίσκεται σε εκκρεμότητα γίνεται **όταν** αυτό ξεμπλοκαριστεί

- Τα `SIGKILL` και `SIGSTOP` **δεν** μπλοκάρονται

Λειτουργίες συνόλων σημάτων

- `sigset_t`
  - σύνολο σημάτων
- `int sigemptyset(sigset_t *set)`
  - αρχικοποίηση κενού συνόλου
- `int sigfillset(sigset_t *set)`
  - αρχικοποίηση γεμάτου συνόλου
- `int sigaddset(sigset_t *set, int sig)`
  - προσθήκη σήματος στο σύνολο
- `int sigdelset(sigset_t *set, int sig)`
  - αφαίρεση σήματος από το σύνολο
- `int sigismember(const sigset_t *set,int sig)`
  - έλεγχος για το αν το σήμα ανήκει στο σύνολο

Καθορισμός μάσκας μπλοκαρίσματος

```
int sigprocmask(int how,
                const sigset_t *set,
                sigset_t *oset);
```

- Η `how` προσδιορίζει τον τρόπο αλλαγής της μάσκας μπλοκαρίσματος, με βάση το σύνολο σημάτων `set`
  - `SIG_BLOCK`: μπλοκάρισμα των σημάτων του `set`
  - `SIG_UNBLOCK`: ξεμπλοκάρισμα των σημάτων του `set`
  - `SIG_SETMASK`: αντικατάσταση της μάσκας με το `set`
- Στο `oset` αποθηκεύεται η παλιά μάσκα μπλοκαρίσματος
  - για να μπορεί να αποκατασταθεί
- Όταν ξεμπλοκαριστεί ένα σήμα που βρίσκεται σε εκκρεμότητα, γίνεται και ο **χειρισμός** του σήματος

Εξέταση εκκρεμών σημάτων

```
int sigpending(sigset_t *set);
```

- Αποθηκεύει στο `set` τα σήματα που έχουν **σταλεί** στην διεργασία και βρίσκονται σε **εκκρεμότητα**
- **Δεν** επιστρέφεται πληροφορία για το πόσες φορές κάθε ένα από αυτά έχει σταλεί στην διεργασία ενώ ήταν μπλοκαρισμένο

- Το λειτουργικό **δεν** καταγράφει πόσες φορές έχει σταλεί ένα μπλοκαρισμένο σήμα σε μια διεργασία **ούτε** γίνεται επανειλημμένος χειρισμός του σήματος όταν αυτό ξεμπλοκαριστεί

```
static void myhandler(int sig) {
  write(STDOUT_FILENO,"got SIGINT\n",11);
}

int main(int argc, char *argv[]) {
  struct sigaction action = {{0}};
  sigset_t s1, s2;

  action.sa_handler = myhandler;
  sigaction(SIGINT, &action, NULL);

  sigemptyset(&s1); sigaddset(&s1, SIGINT);

  while (1) {
    sigprocmask(SIG_BLOCK, &s1, NULL); // block SIGINT
    sleep(10);
    sigpending(&s2);
    if (sigismember(&s2, SIGINT)) {
      printf("SIGINT is pending\n");
    }
    sigprocmask(SIG_UNBLOCK, &s1, NULL); // unblock SIGINT
    sleep(10);
  }
  return(0);
}
```

Μπλοκάρισμα και παράβλεψη σήματος

- Το μπλοκάρισμα ενός σήματος έχει νόημα **μόνο** αν το πρόγραμμα επιθυμεί «τελικά» να παραλάβει και να χειριστεί το σήμα (ξεμπλοκάροντας το)

- Αν η παραλαβή / χειρισμός ενός σήματος είναι γενικότερα **ανεπιθύμητα**, είναι πιο απλό να εγκατασταθεί για αυτό η ενέργεια παράβλεψης

- Αυτό μπορεί να γίνει εγκαθιστώντας/καθορίζοντας τον ειδικό χειριστή σήματος `SIG_IGN` έτσι ώστε το σήμα να αγνοείται αυτόματα (από το ΛΣ)

Αλλαγή μάσκας και αναμονή σήματος

```
int sigsuspend(const sigset_t *sigmask);
```

- Αντικαθιστά την μάσκα μπλοκαρίσματος με το σύνολο σημάτων `sigmask`
- **Και** ταυτόχρονα μπλοκάρει μέχρι να παραληφθεί ένα **μη μπλοκαρισμένο** σήμα για το οποίο υπάρχει ρητός **χειρισμός** από το πρόγραμμα

- Όταν παραληφθεί ένα σήμα, αφού γίνει χειρισμός του σήματος, **αποκαθίσταται** η παλιά μάσκα, και η κλήση επιστρέφει με `-1` και `errno` ίση με `EINTR`

Αναμονή για εκκρεμή σήματα

```
int sigwait(const sigset_t *set,
            int *signum);
```

- Μπλοκάρει μέχρι κάποιο από τα σήματα στο σύνολο `set` να βρεθεί σε **εκκρεμότητα**
- Ένα από τα εκκρεμή σήματα γίνεται **αποδεκτό** και αποθηκεύεται στην `signum`
  - αν υπάρχουν περισσότερα εκκρεμή σήματα, επιστρέφεται ένα από αυτά (απροσδιόριστη επιλογή)
- **Δεν** γίνεται χειρισμός του σήματος
  - παρόλα αυτά, το σήμα θεωρείται **παραδομένο**
- Ελεγχόμενη παραλαβή και παράβλεψη ενός σήματος **χωρίς** την εγκατάσταση κάποιου χειριστή για αυτό

```
static void myhandler(int sig) {
  write(STDOUT_FILENO,"got SIGINT\n",11);
}

int main(int argc, char *argv[]) {
  struct sigaction action = {{0}};
  sigset_t s1, s2;

  action.sa_handler = myhandler;
  sigaction(SIGINT, &action, NULL);

  sigemptyset(&s1); sigaddset(&s1, SIGINT);

  while (1) {
    sigprocmask(SIG_BLOCK, &s1, NULL); // block SIGINT
    sleep(10);
    sigpending(&s2);
    if (sigismember(&s2, SIGINT)) {
      printf("SIGINT is pending\n");
     sigwait(&s1,&sig);
    }
    sigprocmask(SIG_UNBLOCK, &s1, NULL); // unblock SIGINT
    sleep(10);
  }
  return(0);
}
```

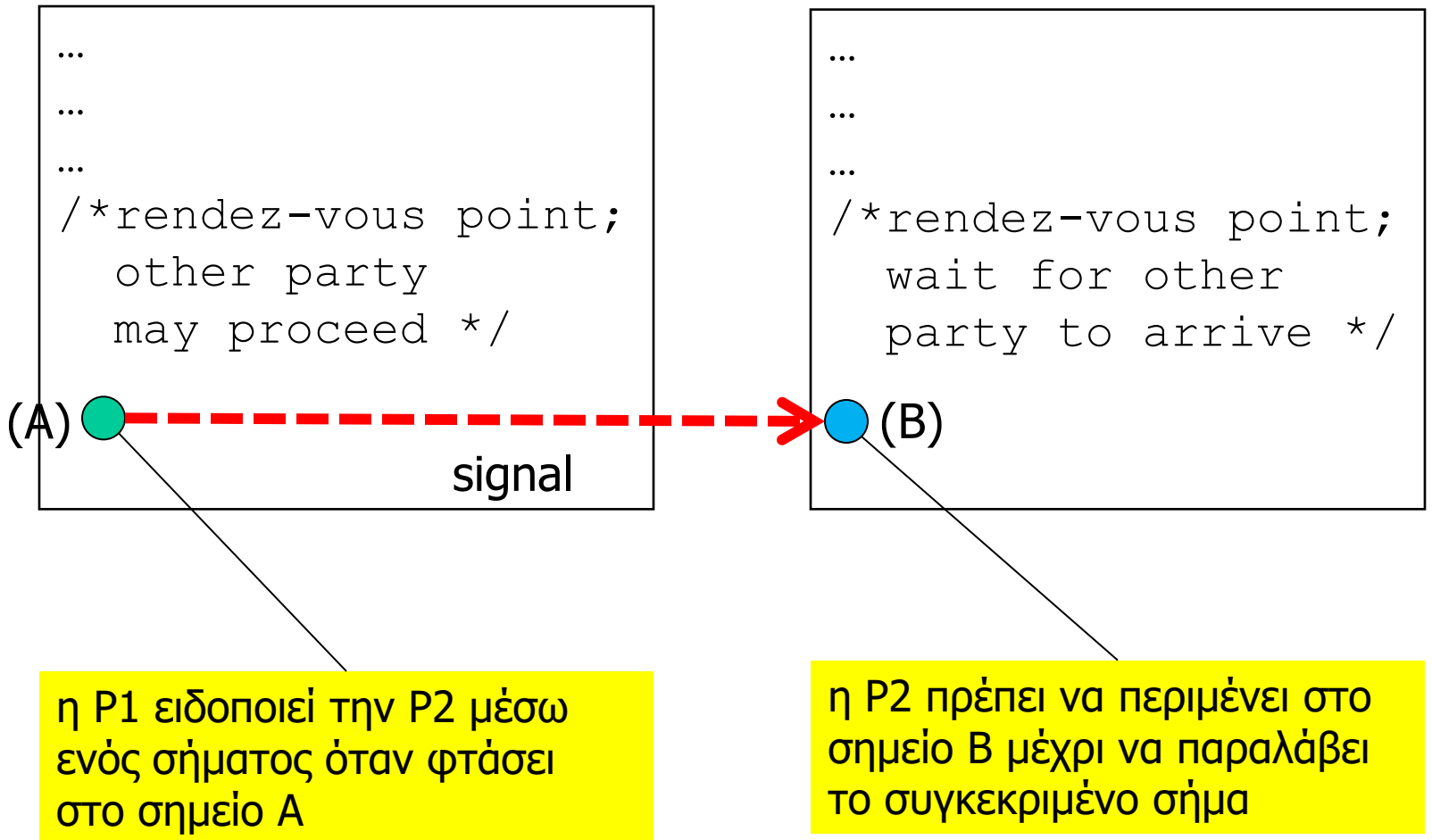
πως θα αλλάξει τώρα η συμπεριφορά του προγράμματος;

Συγχρονισμός διεργασιών με σήματα

- Τα σήματα μπορεί να χρησιμοποιηθούν για να υλοποιηθεί **συγχρονισμός** μεταξύ διεργασιών

- Παράδειγμα: **ασύμμετρο ραντεβού**
- Η διεργασία P2 πρέπει να **περιμένει** προτού συνεχίσει την εκτέλεση της, **μέχρι** η διεργασία P1 να ολοκληρώσει μια διαδικασία (τα αποτελέσματα της οποίας πιθανώς επηρεάζουν την διεργασία P2)

- **Υλοποίηση με σήματα:** η P1 στέλνει σήμα στην P2, και η P2 περιμένει να λάβει το σήμα για να συνεχίσει την εκτέλεση της


…
…
…
/*rendez-vous point;
other party
may proceed */
(A)
signal
…
…
…
/*rendez-vous point;
wait for other
party to arrive */
(B)
η P1 ειδοποιεί την P2 μέσω ενός σήματος όταν φτάσει στο σημείο A
η P2 πρέπει να περιμένει στο σημείο B μέχρι να παραλάβει το συγκεκριμένο σήμα

```
int main(int argc, char *argv[]) {
  pid_t pid; sigset_t s; int signum;

  sigemptyset(&s); sigaddset(&s, SIGUSR1);
  sigprocmask(SIG_BLOCK, &s ,NULL);

  if (!(pid=fork())) {
    sigwait(&s, &signum); // wait for SIGUSR1
    printf("this should print second\n");
    return(0);
  }

  sleep(1);
  printf("this should print first\n");
  kill(pid, SIGUSR1);
  return(0);
}
```

Σχεδιασμός χειριστών σημάτων

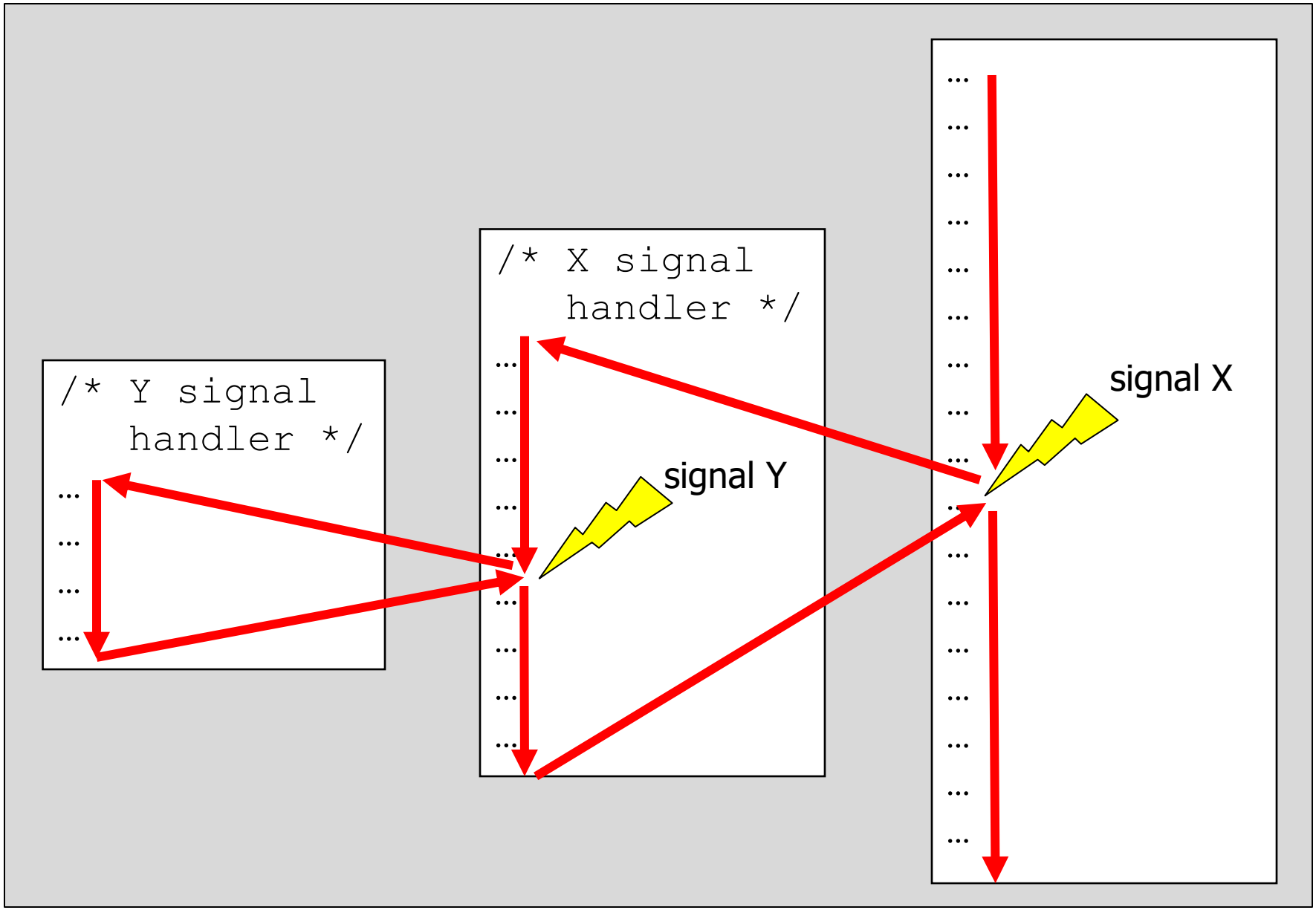
- Ο χειριστής ενός ασύγχρονου σήματος μπορεί να εκτελεστεί ανά πάσα στιγμή, **διακόπτοντας** την κανονική ροή εκτέλεσης του προγράμματος

- Υπάρχουν **συγκεκριμένες** λειτουργίες που είναι **ασφαλείς** για (ασύγχρονα) σήματα
  - μπορούν να χρησιμοποιηθούν μέσα σε χειριστές σημάτων

- Αν ένας χειριστής σήματος χρησιμοποιεί καθολικές μεταβλητές, τότε αυτές πρέπει να δηλώνονται ως `volatile sig_atomic_t`
  - διαφορετικά μπορεί να προκύψουν προβλήματα ασυνέπειας

Διακοπή χειριστή σήματος

- Κατά σύμβαση, το σήμα που προκάλεσε τον χειρισμό σήματος παραμένει **μπλοκαρισμένο** κατά την εκτέλεση του αντίστοιχου χειριστή σήματος
  - αυτό μπορεί να αλλάξει με κατάλληλα options
- Η εκτέλεση ενός χειριστή σήματος του προγράμματος μπορεί όμως να **διακοπεί** από άλλα σήματα
  - αν προκύψουν κατά την εκτέλεση του χειριστή σήματος
- Αν αυτό δεν είναι επιθυμητό, το πρόγραμμα πρέπει να ζητήσει να **μπλοκαριστούν** τα αντίστοιχα σήματα (κατά την εκτέλεση του χειριστή σήματος)
  - βλέπε `sa_mask` της δομής χειρισμού σήματος


/* Y signal
handler */
/* X signal
handler */
signal Y
signal X

Σήματα και `fork` / `exec`

- Η διεργασία παιδί που δημιουργείται μέσω `fork`
  - κληρονομεί **τους χειρισμούς σημάτων** του γονέα
  - κληρονομεί **την μάσκα μπλοκαρίσματος**
  - **δεν** κληρονομεί **τα εκκρεμή σήματα** του γονιού (αρχίζει την εκτέλεση της **χωρίς** εκκρεμή σήματα)

- Όταν μια διεργασία αλλάζει κώδικα μέσω `exec`
  - για σήματα που η καθορισμένη ενέργεια είναι `SIG_DFL`/`SIG_IGN`, στο παιδί αυτή **παραμένει** ως έχει στον γονιό (το `SIGCHLD` μπορεί να γίνει `SIG_DFL` από `SIG_IGN`, ανάλογα με την υλοποίηση)
  - τα σήματα με χειριστή **γυρίζουν** σε `SIG_DFL` – γιατί;
  - η μάσκα μπλοκαρίσματος **παραμένει** ως έχει
  - τα εκκρεμή σήματα **παραμένουν** ως έχουν

Ξυπνητήρια

```
int setitimer(int which,
              const struct itimerval *nval,
              struct itimerval *oval);
```

- `ITIMER_REAL`: μετράει τον πραγματικό χρόνο, στην λήξη στέλνεται το σήμα `SIGALRM`
- `ITIMER_VIRTUAL`: μετράει τον χρόνο εκτέλεσης της διεργασίας, στην λήξη στέλνεται `SIGVTALRM`
- `ITIMER_PROF`: μετράει τον χρόνο εκτέλεσης της διεργασίας και τον χρόνο εκτέλεσης του συστήματος για λογαριασμό της διεργασίας, στην λήξη στέλνεται το σήμα `SIGPROF`
- Στο `oval` επιστρέφονται οι τρέχουσες ρυθμίσεις

Δομή χρονισμού ξυπνητηριού

```
struct itimerval {
   struct timeval it_interval; /*nxt val*/
   struct timeval it_value;    /*cur val*/
};
struct timeval {
   time_t tv_sec;         /* seconds */
   suseconds_t tv_usec; /* microseconds */
};
```

- Το πεδίο `it_value` καθορίζει/επιστρέφει τον χρόνο που απομένει, και το πεδίο `it_interval` την τιμή επαναφοράς για τις επόμενες επαναλήψεις
  - αν δοθούν μηδενικές τιμές, το ξυπνητήρι σταματάει

```
static void myhandler(int sig) {
  write(STDOUT_FILENO,"got SIGALRM\n",12);
}

int main(int argc, char *argv[]) {
  int n1, n2;
  struct sigaction action = { {0} };
  struct itimerval t = { {0} };

  n1 = atoi(argv[1]);
  n2 = atoi(argv[2]);

  action.sa_handler = myhandler;
  sigaction(SIGALRM,&action,NULL);

  t.it_value.tv_sec = n1;    /* first tick */
  t.it_value.tv_usec = 0;
  t.it_interval.tv_sec = n2; /* next ticks */
  t.it_interval.tv_usec = 0;

  setitimer(ITIMER_REAL,&t,NULL);

  while (1) {
    printf("ping\n");
    sleep(10);
  }

  return(0);
}
```

Απλό ξυπνητήρι

- Αν διεργασία επιθυμεί να θέσει ένα απλό ξυπνητήρι για να λάβει το σήμα `SIGALRM`, μπορεί να το κάνει μέσω της

```
unsigned int alarm(unsinged int secs);
```

- Θέτει ένα ξυπνητήρι για να χτυπήσει αφού παρέλθει ο χρόνος `secs` (δευτερόλεπτα)
  - το ξυπνητήρι θα χτυπήσει **μόνο** μια φορά
- Αν το `secs` είναι `0` τότε ακυρώνεται το ξυπνητήρι
  - αν αυτό είχε τεθεί προηγουμένως
- Η τιμή επιστροφής είναι ο αριθμός δευτερολέπτων που απομένουν μέχρι να χτυπήσει το ξυπνητήρι που είχε τεθεί προηγούμενος (ή `0` αν δεν υπάρχει)

```
static void myhandler(int sig) {
  write(STDOUT_FILENO,"got SIGALRM\n",12);
}

int main(int argc, char *argv[]) {
  int n;
  struct sigaction action = { {0} };

  n = atoi(argv[1]);

  action.sa_handler = myhandler;
  sigaction(SIGALRM,&action,NULL);

  alarm(n);

  while (1) {
    printf("ping\n");
    sleep(10);
  }

  return(0);
}
```