Humboldt University

Computer Science Department
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Operating Systems Principles

Lab 3 – User Level Threads

Lab 2

Gesucht: eine einfache Shell

1. Bereitstellen einer **Kommandozeile** und **Ausführen eines Programms** mit Kommandozeilenparametern
   1. Aktuelles Verzeichnis relativ zum Einstiegspunkt als Prompt
2. Ausführen eines Programms im **Hintergrund** mit **Rückgabe der PID**
3. Wechseln des **Arbeitsverzeichnisses**
4. **Warten** auf Prozesse.
   a) Warten auf eine **Teilmenge** der aktuell ausgeführten Hintergrundprozesse pid1 ... pidN
   b) Das "wait" soll mit Ctrl+C **unterbrechbar** sein
   c) Bei Terminierung: möglichst viel Informationen über den **Endzustand** des Prozesses ausgeben
      - Grund des Terminierens, Rückgabewert, …

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Drittes Praktikum

- 2 Wochen Zeit zum Lösen der Aufgaben
  - Aufgabenstellung auf der SAR Website
  - Abgabe über GOYA

- In dieser Woche
  - Erläutern der Aufgabenstellung
- Nächste Woche
  - Zeit zur Bearbeitung
- Nächste Veranstaltung
  - 24./25./31.05.2012 & 01.06.2012

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Lab 3 explained

- **User Level Thread Library**
  - Unterbrechung nicht an beliebiger Stelle: **nicht-präemptiv**
  - Threads geben die CPU **freiwillig auf** oder **warten** auf das Eintreffen eines Ereignisses
  - Threads und Scheduler teilen sich **einen Prozess**

```
typedef void (*ult_func)();
extern int  ult_spawn(ult_func f);
extern void ult_yield();
extern void ult_exit(int status);
extern int  ult_waitpid(int tid, int *status);
extern int  ult_read(int fd, void *buf, int count);
extern void ult_init(ult_func f);
```


Systems Architecture Group
http://sar.informatik.hu-berlin.de

Lab 3 explained

```
/* type of function executed by thread */
typedef void (*ult_func)();

/* spawn a new thread, return its ID */
extern int ult_spawn(ult_func f);
```

- Erzeugen eines Threads: **`ult_spawn`**
  - Eintrittspunkt **`f`** mit Prototyp **`ult_func`**
  - Erzeugen der internen Strukturen: TCB, etc.
  - Einordnen in die Run-Queue
  - An dieser Stelle kein Scheduling
  - Rückgabe der Thread-ID

Systems Architecture Group
http://sar.informatik.hu-berlin.de

Lab 3 explained

```
/* yield the CPU to another thread */
extern void ult_yield();
```

- Freiwillige Aufgabe des Prozessors
  - Aktivieren des Schedulers
  - Auswahl eines anderen, lauffähigen Threads
  - Aktivieren des ausgewählten Threads

Lab 3 explained

```
/* current thread exits */
extern void ult_exit(int status);
```

- Beenden eines User-Threads
  - Der aufrufende Thread wird zum Zombie
  - Speichern des Exit-Status
  - Exit-Status kann mit **`ult_waitpid`** abgefragt werden

Systems Architecture Group
http://sar.informatik.hu-berlin.de

Lab 3 explained

```
/* thread waits for termination of another thread */
/* returns 0 for success, -1 for failure */
/* exit-status of terminated thread is obtained */
extern int ult_waitpid(int tid, int *status);
```

- Warten auf Threads
  - Warten auf Beendigung des Threads mit ID **`tid`**
  - Rufender Thread wird erst nach der Beendigung fortgesetzt
  - Liefert den Exit-Status zurück
  - Ist der Thread schon beendet, dann kehrt der Aufruf sofort zurück
  - Andernfalls muss der Scheduler regelmäßig überprüfen, ob der Thread beendet wurde

Lab 3 explained

```
/* read from file, block the thread if no data available */
extern int ult_read(int fd, void *buf, int count);
```

- Lesen aus Dateien
  - Wrapper-Funktion für die Funktion **`read`** der Unix-API
  - Problem: read ist ein Systemruf und kann im Kernel blockieren
    - Damit blockiert der Prozess und alle User-Threads
  - Lösung: Vorheriges Überprüfen, ob ausreichend Daten vorhanden sind
    - Wenn nicht, blockiert der rufende Thread und gibt die CPU an einen Anderen ab
    - Scheduler muss wieder regelmäßig überprüfen, ob der rufende Thread fortgesetzt werden kann
    - Realisierung: Non-blocking IO

Lab 3 explained

```
/* start scheduling and spawn a thread running function f */
extern void ult_init(ult_func f);
```

- Initialisieren der Thread-Bibliothek
  - Muss als erste Funktion der Bibliothek vor allen anderen gerufen werden
  - Anlegen der Datenstrukturen für Scheduler und Thread-Verwaltung
  - Erzeugen eines initialen Threads
  - Warten auf Beendigung des initialen Threads

ULT Example

```
void threadA() {…}
void threadB() {…}
void myInit()  {…}

int  main()
{
    printf("starting myInit\n"); fflush(stdout);
    ult_init(myInit);
    exit(0);
}
```

ULT Example

```
void myInit()  {
    int cpid[2], i, status;

    cpid[0] = ult_spawn(threadA);

    cpid[1] = ult_spawn(threadB);

    for (i = 0; i < 2; i++) {
        if (ult_waitpid(cpid[i], &status) == -1) {
            ult_exit(-1);
        }
    }
    ult_exit(0);
}
```

Lab 3 explained

Test-Programm

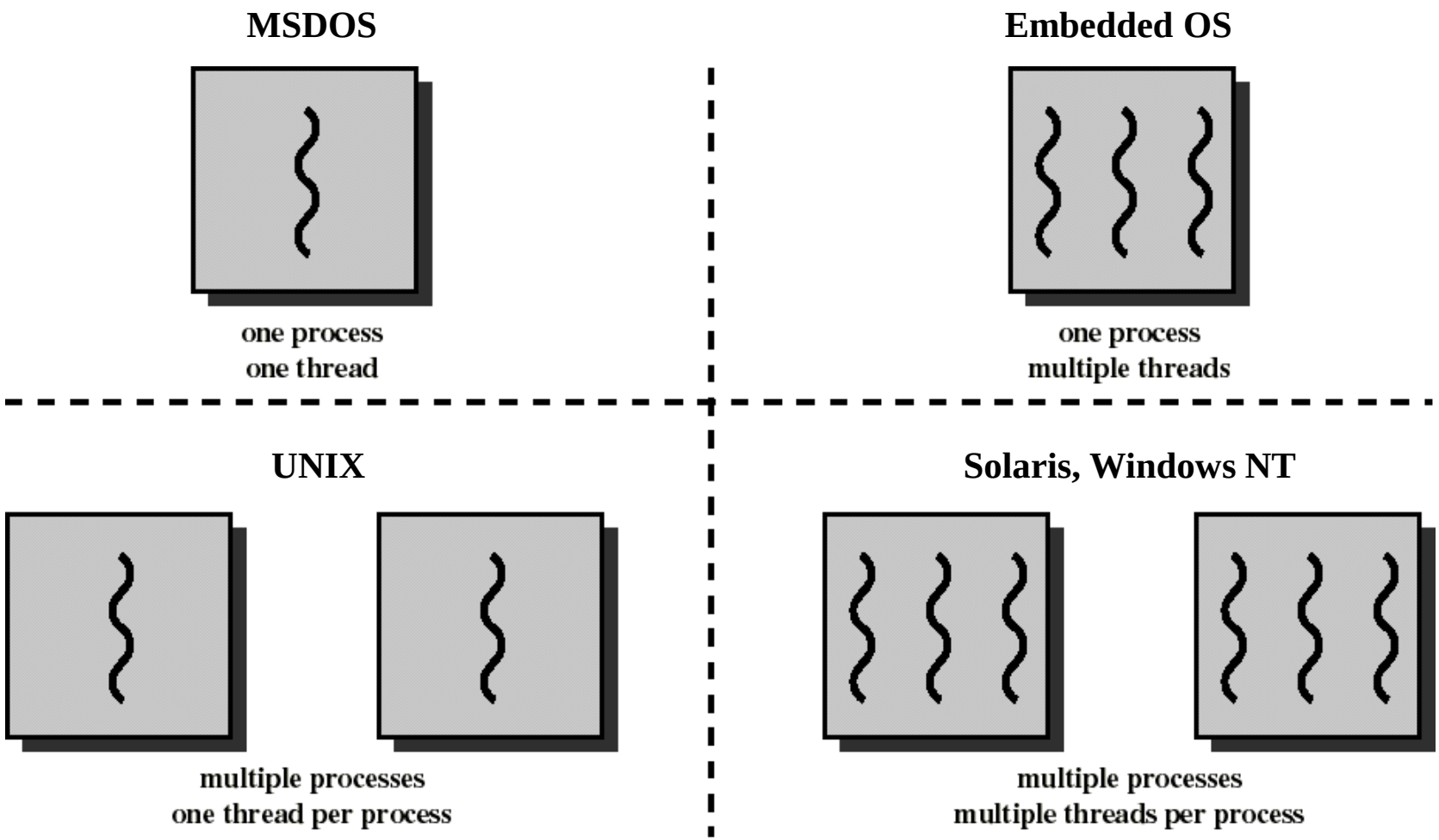
- Thread A kopiert aus **/dev/random** nach **/dev/null**
  - Führt diverse Statistiken mit
    - Anzahl an bereits kopierten Bytes
    - Zeit seit Start des Threads
    - Durchsatz
    - …
- Thread B stellt eine Shell bereit
  - Auslesen der Statistiken des Threads A (Kommando **stats**)
  - Beenden des Programms (Kommando **exit**)

```
/* type of function executed by thread */
typedef void (*ult_func)();

/* spawn a new thread, return its ID */
extern int ult_spawn(ult_func f);

/* yield the CPU to another thread */
extern void ult_yield();

/* current thread exits */
extern void ult_exit(int status);

/* thread waits for termination of another thread */
/* returns 0 for success, -1 for failure */
/* exit-status of terminated thread is obtained */
extern int ult_waitpid(int tid, int *status);

/* read from file, block the thread if no data available */
extern int ult_read(int fd, void *buf, int count);

/* start scheduling and spawn a thread running function f */
extern void ult_init(ult_func f);
```

Processes vs. Threads

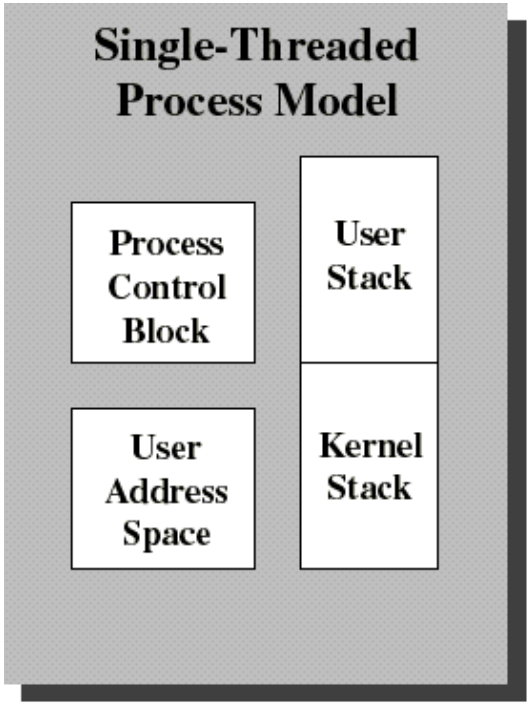
- Process
  - Independent memory space
  - Protected access to other resources
  - Large context
  - Heavy weight: (expensive to create and switch between)
- Threads
  - Shared memory space with other threads
  - Shared resources (i.e. unprotected)
  - Small context
  - Light weight: (efficient to create and switch between)

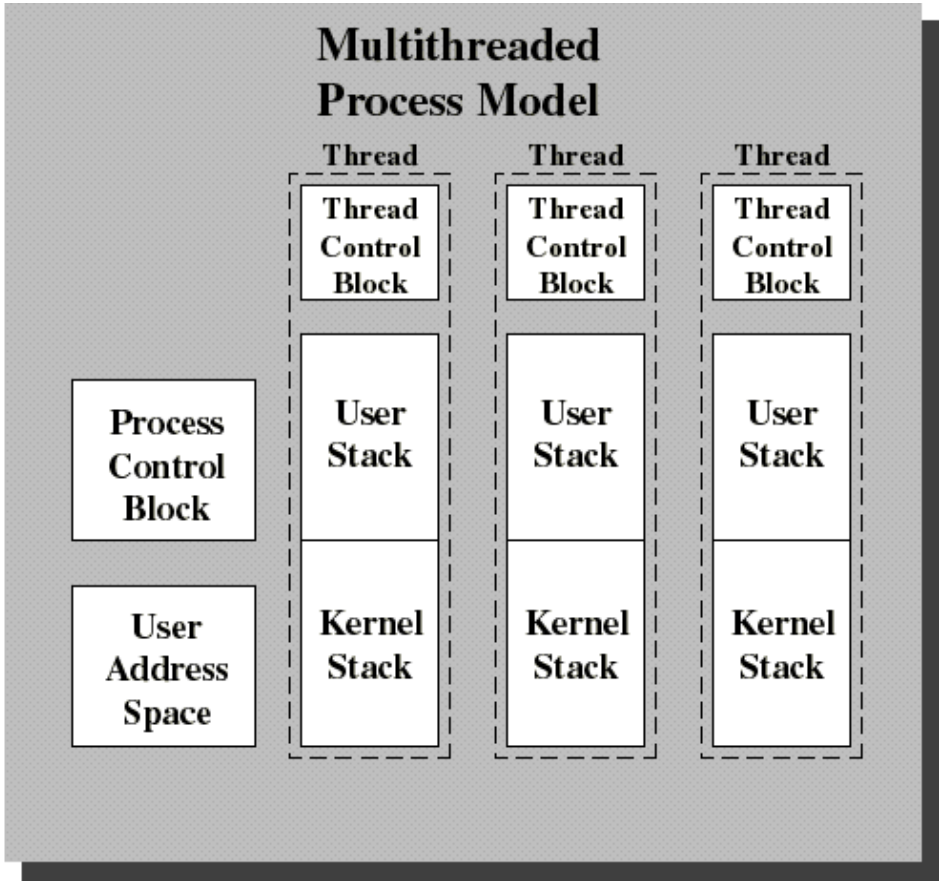
OS Support for Processes / Threads



Systems Architecture Group
http://sar.informatik.hu-berlin.de

Allocation of Process State





Thread Control Block contains a register image, thread priority and thread state information

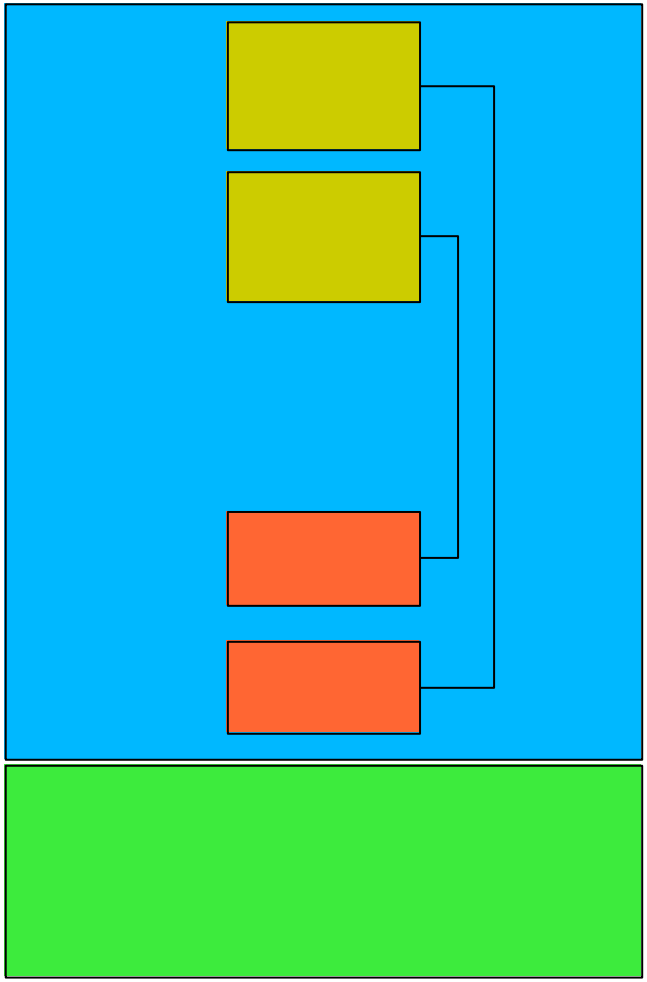
Thread State/Control Information

- Thread Control Block (TCB)
  - Each thread has its own execution state
  - A thread has a small amount of local data
    - Local stack
  - All threads share the global address space of their process
    - Data written by one thread is seen by all others
    - What problems are caused by threads sharing local data??

- Thread Queues
  - Separate queues for running, waiting, zombie, etc.

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Big Picture



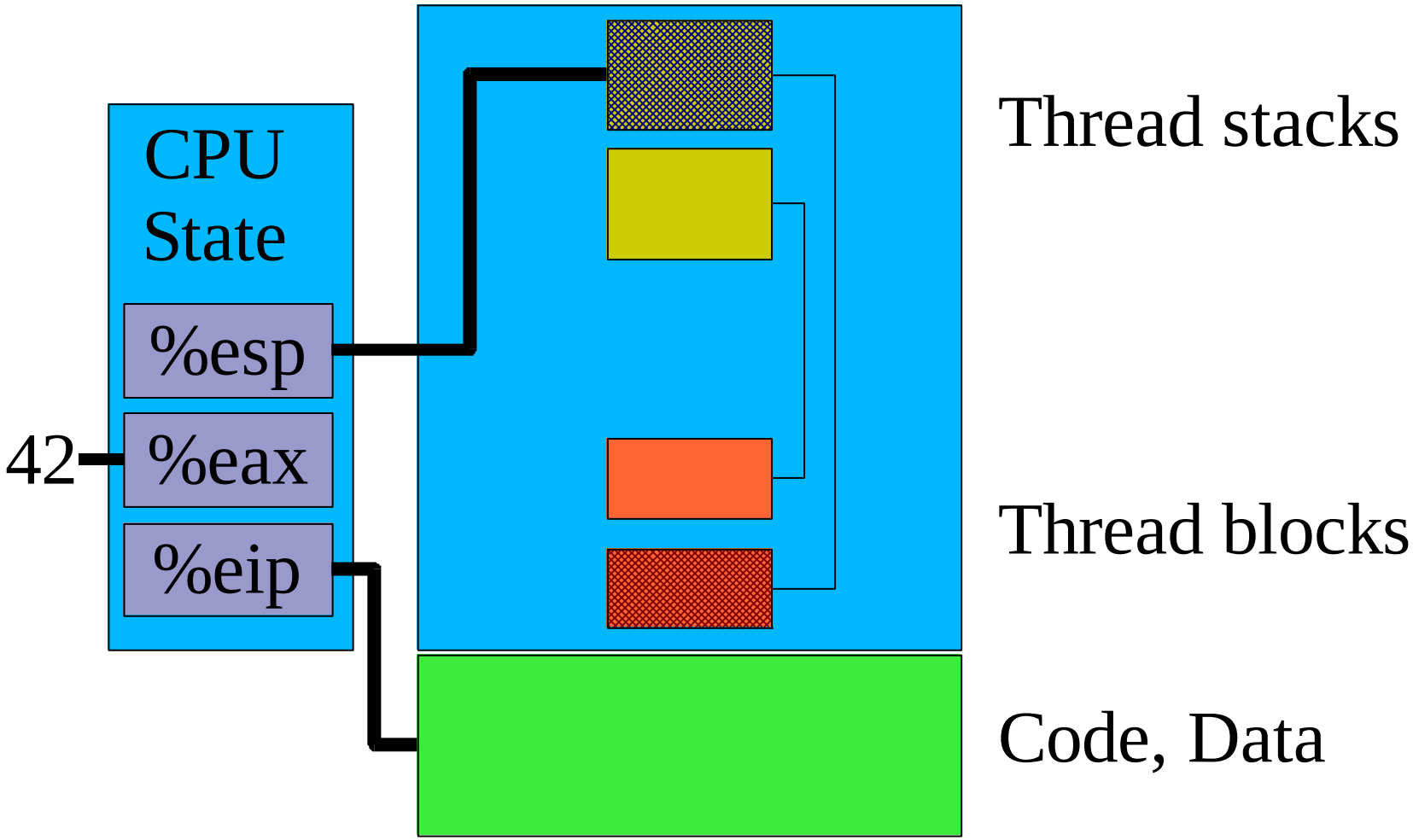
Thread stacks

Thread blocks

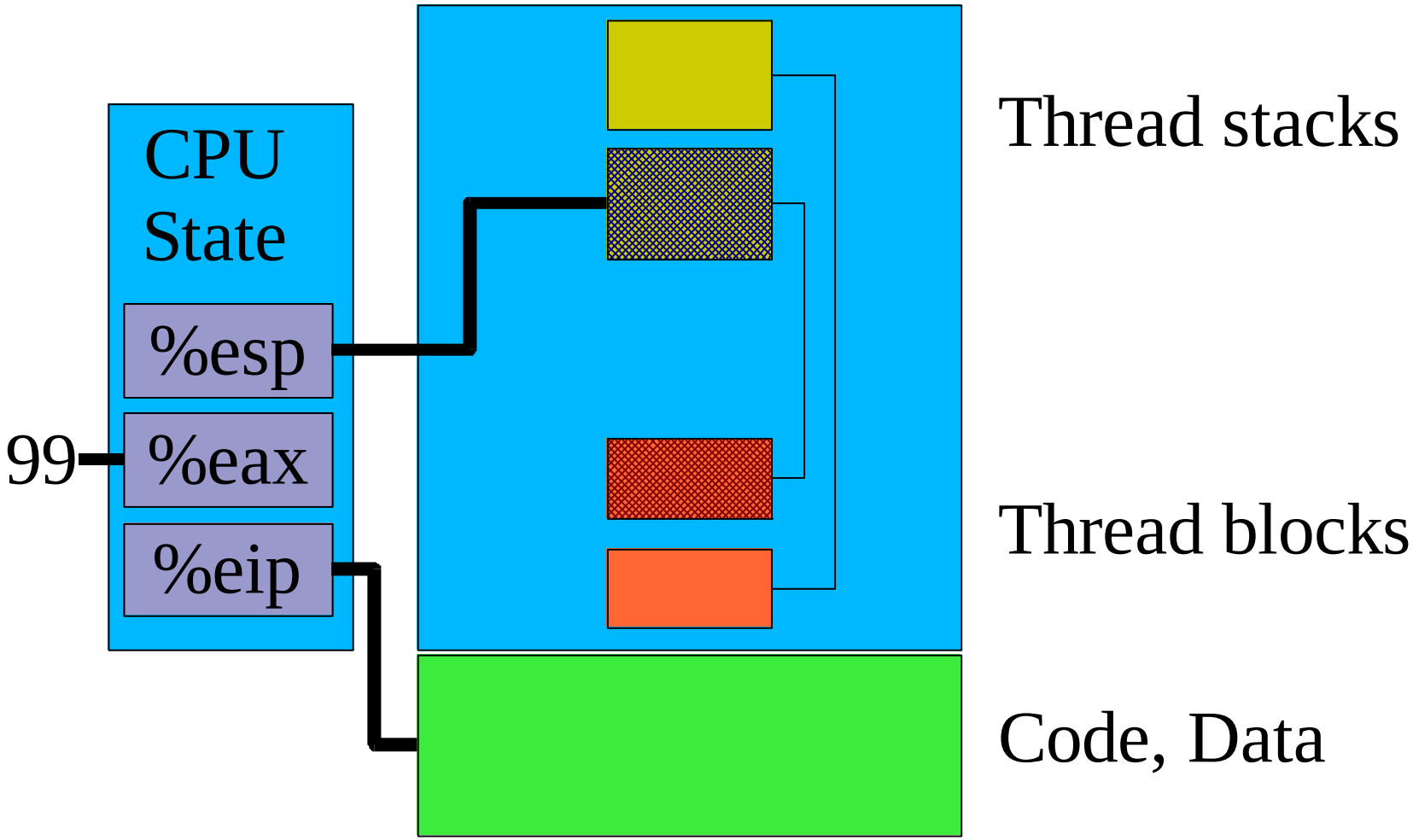
Code, Data

Systems Architecture Group
http://sar.informatik.hu-berlin.de

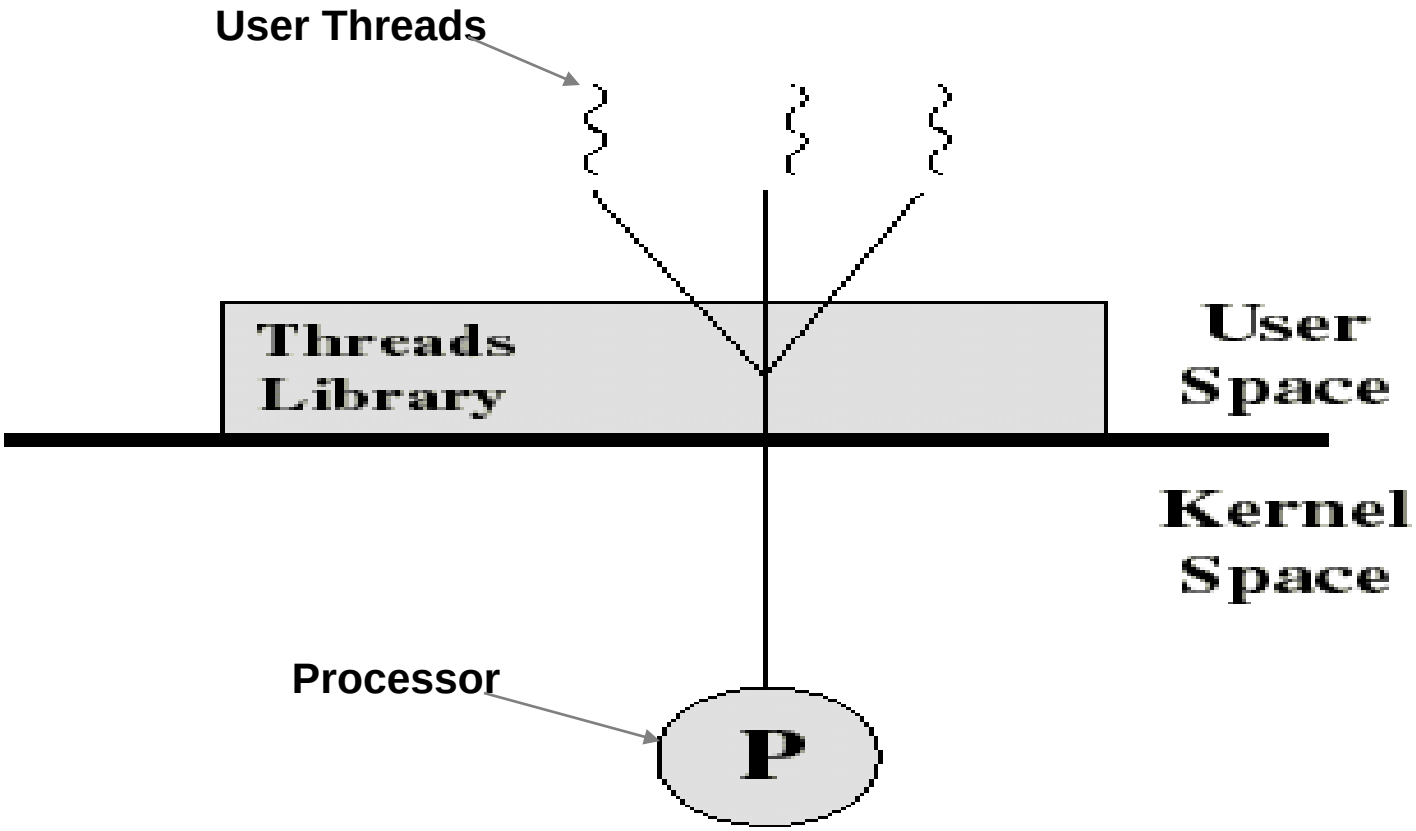
Big Picture


CPU
State
%esp
42
%eax
%eip
Thread stacks
Thread blocks
Code, Data

Big Picture


CPU
State
%esp
99
%eax
%eip
Thread stacks
Thread blocks
Code, Data

User Level Threads (ULT)



User Level Threads (ULTs)

- ULTs are implemented in an application library
  - Creation, termination, and communication
  - Scheduling and context switching
- The kernel is not aware of the threads
- The kernel only schedules the process
  - Thread library divides time amongst threads

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

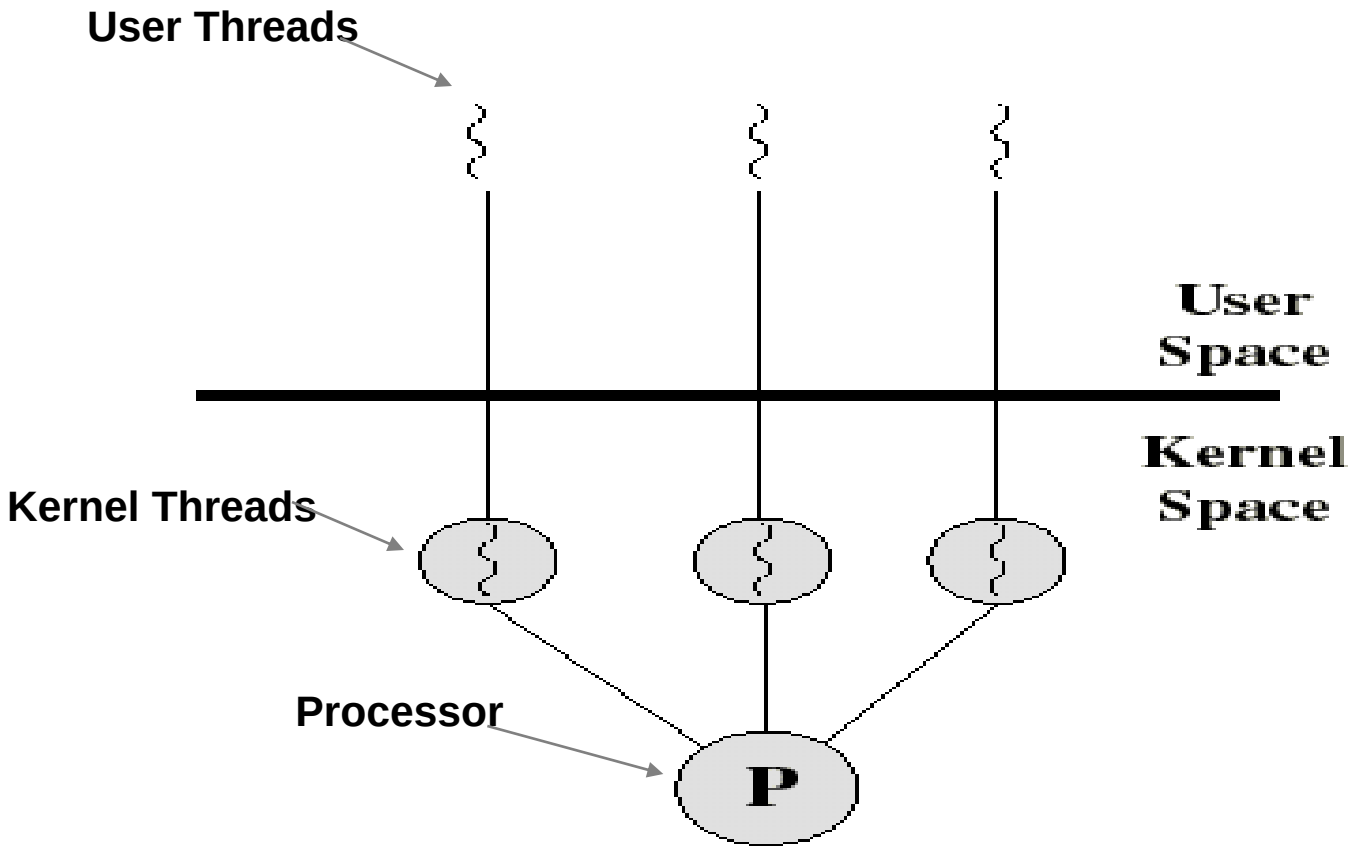
Advantages of ULTs

- Switching between threads is fast
  - Requires a mode switch not a process switch
- Thread library can run on any platform
  - Portable, ULTs are implemented in software
- Application specific scheduling
  - Each process can use a different scheduling policy (I.e. round robin vs. FIFO)

Disadvantages of ULTs

- A blocking system call in one thread will block all threads
  - The OS does not know that there is more than one thread of execution
- A multi-threaded application cannot take advantage of multi-processing
- No system level priority
  - Priority of threads is based on the priority of the process

Kernel Level Threads (KLTs)



Kernel Level Threads

- All thread management is done by kernel
  - No thread library
  - Each user thread is mapped to a thread in kernel space
  - Kernel maintains context information for the process and the threads
  - Switching between threads requires the kernel
- Scheduling on a thread basis

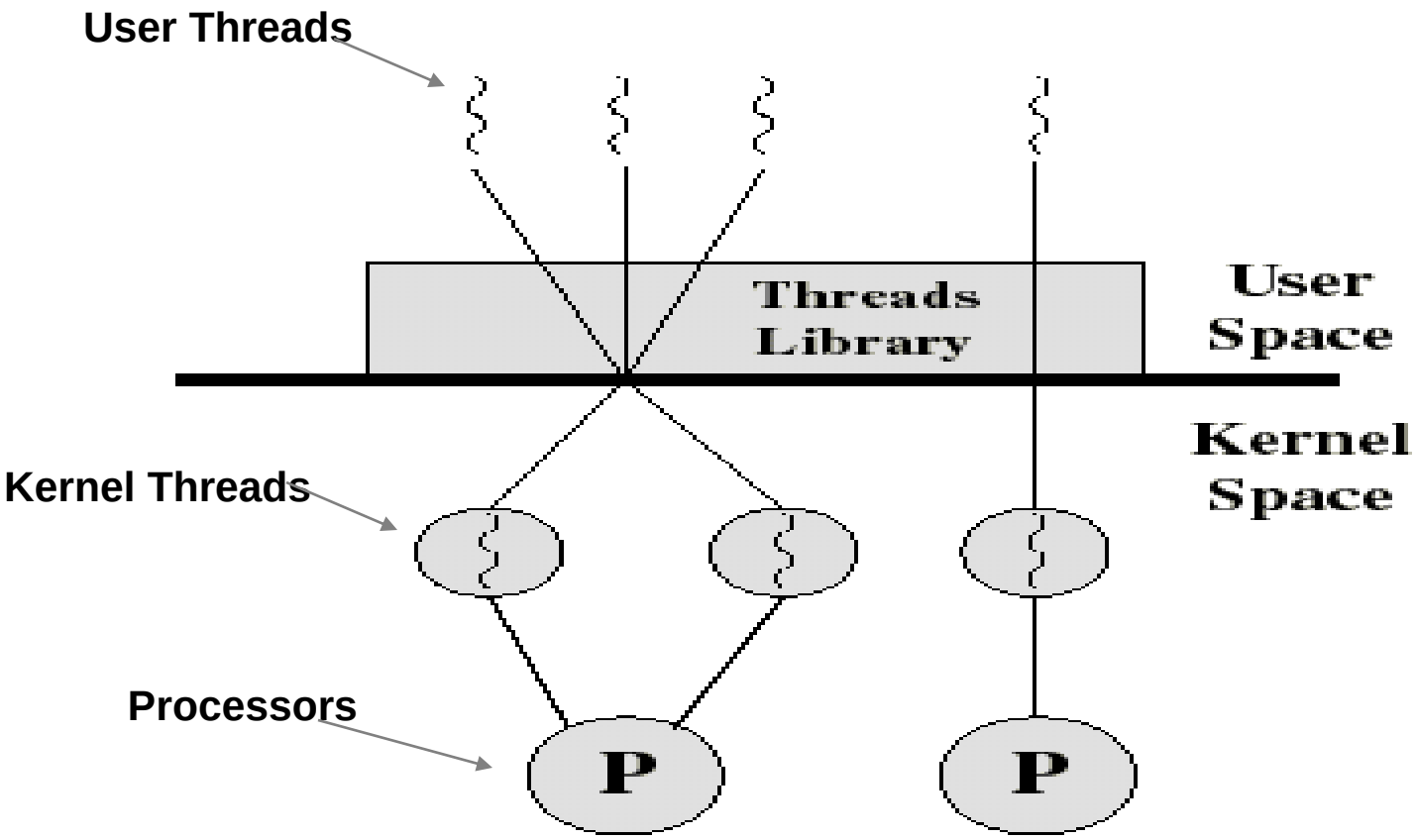
Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Advantages/Disadvantages of KLTs

- Advantages
  - Multiple threads can be scheduled on multiple processors
  - Blocking of one thread does not block others
  - Kernel routines can be multithreaded

- Disadvantages
  - Thread switching must be done by kernel
  - More overhead switching between threads

Combined ULT and KLTs



Systems Architecture Group
http://sar.informatik.hu-berlin.de

Combined ULT/KLT Approaches

- Combines the advantages of both approaches
  - Thread creation done in user space
  - Most scheduling and communication done in user space
  - KLTs run in parallel and do not block other KLTs
- User can adjust the number of KLTs
- Mapping between UL/KL threads can be:
  - 1 to 1 (same as kernel threads alone)
  - M to N (some user threads share)

Slow System Calls

- Ones that can “block” forever
  - i.e., a read from a pipe, terminal, or network device
  - Write when …
  - Open blocks
    - Modem answers phone
    - Network device
    - FIFO for writing when no other process open for reading
  - Reads/writes with mandatory record locks on files

Non-blocking I/O

- Calling open with the flag O_NONBLOCK sets nonblocking attribute for the file_descriptor returned
  - fd = open(path, 0644 | O_CREAT | O_NONBLOCK);
- For a file_descriptor that is already open we can use fcntl to set it to nonblocking

```
#include <fcntl.h>
void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
        int val;
        if ((val = fcntl(fd, F_GETFL, 0)) < 0)
                err_sys("fcntl F_GETFL error");

        val |= flags;             /* turn on flags */
        if (fcntl(fd, F_SETFL, val) < 0)
                err_sys("fcntl F_SETFL error");
}
```

Non-Blocking write

```
#include <errno.h>
#include <fcntl.h>
char    buf[500000];
int main(void)
{
        int ntowrite, nwrite; char    *ptr;
        ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
        fprintf(stderr, "read %d bytes\n", ntowrite);
        set_fl(STDOUT_FILENO, O_NONBLOCK);
        ptr = buf;
        while (ntowrite > 0) {
                errno = 0;
                nwrite = write(STDOUT_FILENO, ptr, ntowrite);
                printf("nwrite = %d, errno = %d\n", nwrite, errno);
                if (nwrite > 0) {
                        ptr += nwrite;
                        ntowrite -= nwrite;
                }
        }
        clr_fl(STDOUT_FILENO, O_NONBLOCK);    /* clear nonblocking */
        exit(0);
}
```

Non-Blocking write

```
$ a.out < /etc/termcap 2>stderr.out
```

output to terminal
lots of output to terminal ...

```
$ cat stderr.out
read 100000 bytes
nwrite = 8192, errno = 0
nwrite = 8192, errno = 0
nwrite = -1, errno = 11
. . .
nwrite = 4096, errno = 0
nwrite = -1, errno = 11
. . .
nwrite = 4096, errno = 0
nwrite = -1, errno = 11
. . .
nwrite = 4096, errno = 0
nwrite = -1, errno = 11
. . .
nwrite = -1, errno = 11
. . .
nwrite = 4096, errno = 0
```

211 of these errors

658 of these errors

604 of these errors

1047 of these errors

1046 of these errors

and so on ...

Systems Architecture Group
http://sar.informatik.hu-berlin.de

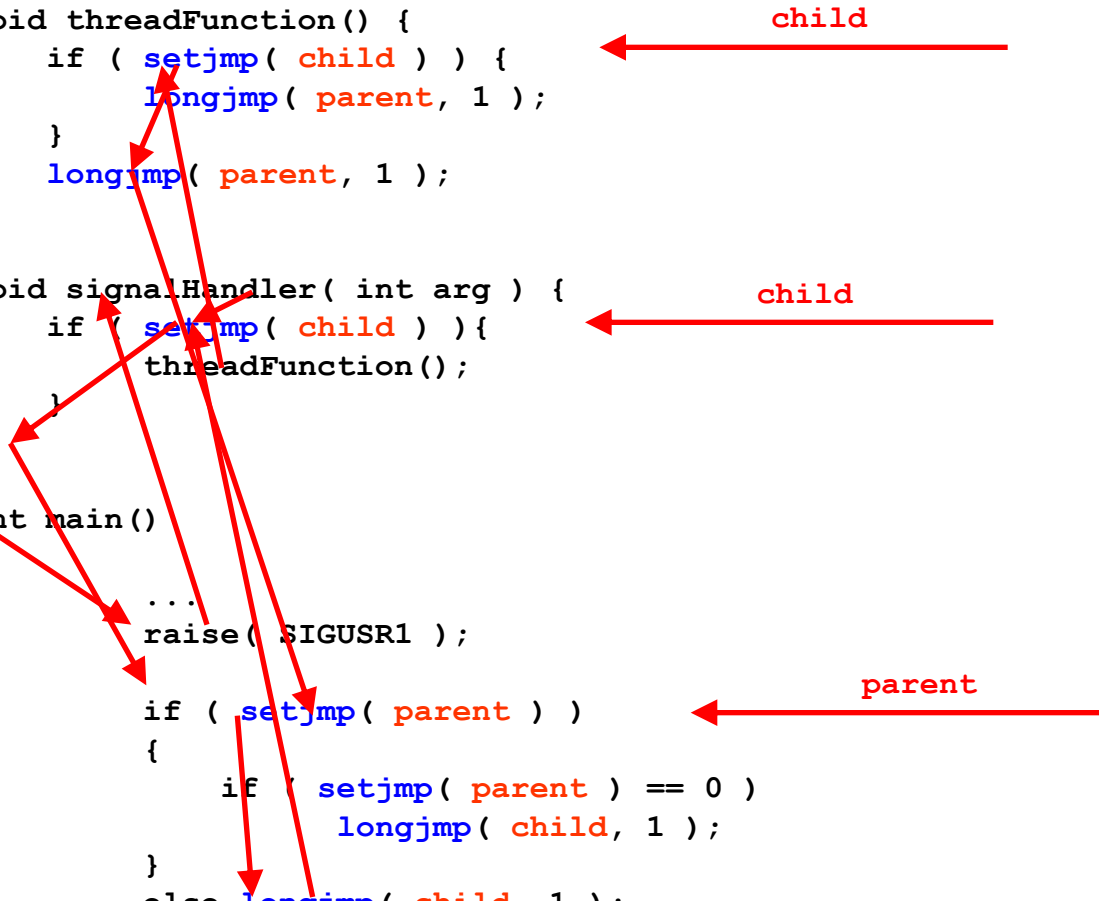
Non-local Jumps (1)

```
void signalHandler( int arg ) {…}

int main()
{
        stack_t stack;
        stack.ss_flags = 0;
        stack.ss_size = STACK;
        stack.ss_sp = malloc(STACK);
        sigaltstack( &stack, 0 );

        struct sigaction sa;
        sa.sa_handler = &signalHandler;
        sa.sa_flags = SA_ONSTACK;
        sigemptyset( &sa.sa_mask );
        sigaction( SIGUSR1, &sa, 0 );

        raise( SIGUSR1 );

        ...

        free( stack.ss_sp );
        return 0;
}
```


Systems Architecture Group
http://sar.informatik.hu-berlin.de

Non-local Jumps (1)

```
jmp_buf child, parent;

void threadFunction() {
    if ( setjmp( child ) ) {
        longjmp( parent, 1 );
    }
    longjmp( parent, 1 );
}

void signalHandler( int arg ) {
    if ( setjmp( child ) ){
        threadFunction();
    }
}

int main()
{
        ...
        raise( SIGUSR1 );

        if ( setjmp( parent ) )
        {
            if ( setjmp( parent ) == 0 )
                longjmp( child, 1 );
        }
        else longjmp( child, 1 );

        return 0;
}
```



Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Misc

- Useful functions:
  - fnctl
  - select
  - open
  - close
  - setjmp
  - longjmp

- The Single UNIX Specification, http://www.opengroup.org/onlinepubs/009695399/