

```
int main(int argc, char** argv) {
    // ...
    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(8080);
    // ...
    struct sockaddr_in client_addr;
    client_addr.sin_family = AF_INET;
    client_addr.sin_port = htons(8080);
    // ...
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("socket");
        return -1;
    }
    // ...
    struct sigaction sa;
    sa.sa_handler = sig_handler;
    sa.sa_flags = SA_SIGINFO;
    sigaction(SIGCHLD, &sa, NULL);
    // ...
    pthread_t pthread;
    pthread_create(&pthread, NULL, pthread_func, &sock);
    // ...
    pthread_join(pthread, NULL);
    // ...
    close(sock);
    return 0;
}
```

```
void sig_handler(int sig, siginfo_t* info, void* ucontext_t) {
    // ...
    struct sigaction sa;
    sa.sa_handler = sig_handler;
    sa.sa_flags = SA_SIGINFO;
    sigaction(SIGCHLD, &sa, NULL);
    // ...
    pthread_t pthread;
    pthread_create(&pthread, NULL, pthread_func, &sock);
    // ...
    pthread_join(pthread, NULL);
    // ...
    close(sock);
    return 0;
}
```

```
void pthread_func(void* arg) {
    int sock = *(int*)arg;
    // ...
    struct sockaddr_in client_addr;
    client_addr.sin_family = AF_INET;
    client_addr.sin_port = htons(8080);
    // ...
    int client_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (client_sock < 0) {
        perror("client socket");
        return;
    }
    // ...
    if (connect(client_sock, &client_addr, sizeof(client_addr)) < 0) {
        perror("connect");
        return;
    }
    // ...
    while (1) {
        int n = read(client_sock, buf, sizeof(buf));
        if (n < 0) {
            perror("read");
            return;
        }
        // ...
        write(sock, buf, n);
    }
}
```

```
int main(int argc, char** argv) {
    // ...
    pthread_t pthread;
    pthread_create(&pthread, NULL, pthread_func, &sock);
    // ...
    pthread_join(pthread, NULL);
    // ...
    close(sock);
    return 0;
}
```

```
int main(int argc, char** argv) {
    // ...
    struct sigaction sa;
    sa.sa_handler = sig_handler;
    sa.sa_flags = SA_SIGINFO;
    sigaction(SIGCHLD, &sa, NULL);
    // ...
    pthread_t pthread;
    pthread_create(&pthread, NULL, pthread_func, &sock);
    // ...
    pthread_join(pthread, NULL);
    // ...
    close(sock);
    return 0;
}
```

```
void pthread_func(void* arg) {
    int sock = *(int*)arg;
    // ...
    struct sockaddr_in client_addr;
    client_addr.sin_family = AF_INET;
    client_addr.sin_port = htons(8080);
    // ...
    int client_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (client_sock < 0) {
        perror("client socket");
        return;
    }
    // ...
    if (connect(client_sock, &client_addr, sizeof(client_addr)) < 0) {
        perror("connect");
        return;
    }
    // ...
    while (1) {
        int n = read(client_sock, buf, sizeof(buf));
        if (n < 0) {
            perror("read");
            return;
        }
        // ...
        write(sock, buf, n);
    }
}
```

```
int main(int argc, char** argv) {
    // ...
    pthread_t pthread;
    pthread_create(&pthread, NULL, pthread_func, &sock);
    // ...
    pthread_join(pthread, NULL);
    // ...
    close(sock);
    return 0;
}
```

```
int main(int argc, char** argv) {
    // ...
    pthread_t pthread;
    pthread_create(&pthread, NULL, pthread_func, &sock);
    // ...
    pthread_join(pthread, NULL);
    // ...
    close(sock);
    return 0;
}
```

```
int main(int argc, char** argv) {
    // ...
    struct sigaction sa;
    sa.sa_handler = sig_handler;
    sa.sa_flags = SA_SIGINFO;
    sigaction(SIGCHLD, &sa, NULL);
    // ...
    pthread_t pthread;
    pthread_create(&pthread, NULL, pthread_func, &sock);
    // ...
    pthread_join(pthread, NULL);
    // ...
    close(sock);
    return 0;
}
```

```
void pthread_func(void* arg) {
    int sock = *(int*)arg;
    // ...
    struct sockaddr_in client_addr;
    client_addr.sin_family = AF_INET;
    client_addr.sin_port = htons(8080);
    // ...
    int client_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (client_sock < 0) {
        perror("client socket");
        return;
    }
    // ...
    if (connect(client_sock, &client_addr, sizeof(client_addr)) < 0) {
        perror("connect");
        return;
    }
    // ...
    while (1) {
        int n = read(client_sock, buf, sizeof(buf));
        if (n < 0) {
            perror("read");
            return;
        }
        // ...
        write(sock, buf, n);
    }
}
```

```
int main(int argc, char** argv) {
    // ...
    pthread_t pthread;
    pthread_create(&pthread, NULL, pthread_func, &sock);
    // ...
    pthread_join(pthread, NULL);
    // ...
    close(sock);
    return 0;
}
```

```
int main(int argc, char** argv) {
    // ...
    pthread_t pthread;
    pthread_create(&pthread, NULL, pthread_func, &sock);
    // ...
    pthread_join(pthread, NULL);
    // ...
    close(sock);
    return 0;
}
```

Stack
 - LIFO (Last In First Out)
 - Speicherbereich für lokale Variablen
 - Funktionsaufrufe
 - Parameterübergabe
 - Rückgabewerte
 - Stackoverflow (wenn Stack voll ist)
 - Stackunderflow (wenn Stack leer ist)

Heap
 - Speicherbereich, den der Programmierer selbst verwalten muss
 - Dynamische Speicherzuweisung
 - Man muss Speicher manuell freigeben
 - Gefahr von Speicherlecks
 - Flexibilität bei der Speichergröße

TCFS → CPU
 - Taktzeit
 - CPU-Zykluszeit
 - Busfrequenz
 - Speicherbandbreite
 - Cache-Größe
 - Cache-Lokalität
 - Cache-Verhalten
 - Cache-Miss-Rate
 - Cache-Refresh-Rate

	FCFS	RR	VRP	SPN	SRTF	HRTN	FB
Wart.	✓	✓	✓	✓	✓	✓	✓
Prob.	✓	✓	✓	✓	✓	✓	✓
St.	✓	✓	✓	✓	✓	✓	✓

- FCFS: First Come First Served
 - RR: Round Robin
 - VRP: Variable Round Robin
 - SPN: Shortest Process Next
 - SRTF: Shortest Remaining Time First
 - HRTN: Highest Response Ratio Next
 - FB: Fairness

Monitor
 - Synchronisationsmechanismus
 - Verhindert gegenseitigen Ausschluss
 - Erlaubt gegenseitigen Ausschluss
 - Verhindert Deadlocks
 - Erlaubt Deadlocks
 - Verhindert Starvation
 - Erlaubt Starvation

Semaphore
 - Zählvariable
 - Erlaubt die Kontrolle der Anzahl von Prozessen in einem kritischen Abschnitt
 - Verhindert Deadlocks
 - Erlaubt Deadlocks
 - Verhindert Starvation
 - Erlaubt Starvation

Stapelverwaltung
 - Stackoverflow vermeiden
 - Stackpointer überwachen
 - Stacklimit setzen
 - Stacksegmentierung

Verwaltung
 - Speicherplatzverwaltung
 - Fragmentierung vermeiden
 - Speicherbanking
 - Speicherbanking vermeiden
 - Speicherbanking nutzen

Validierung
 - Validierung von Variablen
 - Validierung von Zeilen
 - Validierung von Spalten
 - Validierung von Blöcken
 - Validierung von Sätzen

Erweitertes Speicherbanking
 - Erweitertes Speicherbanking
 - Erweitertes Speicherbanking vermeiden
 - Erweitertes Speicherbanking nutzen

Erweitertes Speicherbanking
 - Erweitertes Speicherbanking
 - Erweitertes Speicherbanking vermeiden
 - Erweitertes Speicherbanking nutzen

Lock
 - Thread lokal
 - Signal kommt von sleep
 - Braucht man von awake
 - Systemzustand

Lock
 - Thread lokal
 - Signal kommt von sleep
 - Braucht man von awake
 - Systemzustand

Lock
 - Thread lokal
 - Signal kommt von sleep
 - Braucht man von awake
 - Systemzustand

Lock
 - Thread lokal
 - Signal kommt von sleep
 - Braucht man von awake
 - Systemzustand

Lock
 - Thread lokal
 - Signal kommt von sleep
 - Braucht man von awake
 - Systemzustand

Nachteile Hoch-Sysem
 - Reduziert massive Busbreite
 - Reduziert (worst case) Verteilung
 - Engpass (womöglich) kann nicht proz. auslasten

Nachteile Hoch-Sysem
 - Reduziert massive Busbreite
 - Reduziert (worst case) Verteilung
 - Engpass (womöglich) kann nicht proz. auslasten

Nachteile Hoch-Sysem
 - Reduziert massive Busbreite
 - Reduziert (worst case) Verteilung
 - Engpass (womöglich) kann nicht proz. auslasten

Nachteile Hoch-Sysem
 - Reduziert massive Busbreite
 - Reduziert (worst case) Verteilung
 - Engpass (womöglich) kann nicht proz. auslasten

Nachteile Hoch-Sysem
 - Reduziert massive Busbreite
 - Reduziert (worst case) Verteilung
 - Engpass (womöglich) kann nicht proz. auslasten