Universität Bamberg

EiRBS: Einführung in Rechnersysteme und Betriebssysteme

Vorlesung 12: Ressourcen und Synchronisation

Literatur:
Arpaci-Dusseau [1]
Kap. 7 und 53–55

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insbes. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

Nochmal: schnelle Prozesserzeugung

- Kopieren des Adressraums benötigt viel Zeit
  - Besonders, wenn direkt nach `fork()` dann `exec..()` aufgerufen wird
    ➛ ***komplette Zeitverschwendung!***
- Historische Lösung: `vfork`
  - Der Elternprozess wird suspendiert, bis der Kindprozess `exec..()` aufruft oder mit `_exit()` terminiert
- Der Kindprozess verwendet einfach direkt Code und Daten seines Elternprozesses (keine Kopie!)
  - Dabei darf der Kindprozess ***keine Daten ändern!***
  - Manchmal nicht so einfach: z.B. nicht `exit()`, sondern `_exit()` aufrufen
- Moderne Lösung: ***copy on write***
  - Eltern- und Kindprozess ***nutzen die selben Code- und Datensegmente*** mit Hilfe von virtuellem Speicher und der *memory management unit* (MMU)
  - Ein Segment wird nur kopiert, wenn der Kindprozess Daten darin ändert
  - **Nicht** der Fall, wenn `exec..()` direkt nach `fork()` aufgerufen wird
  - `fork()` mit *copy on write* ist *fast* so schnell wie `vfork()`

Geht das nicht besser?

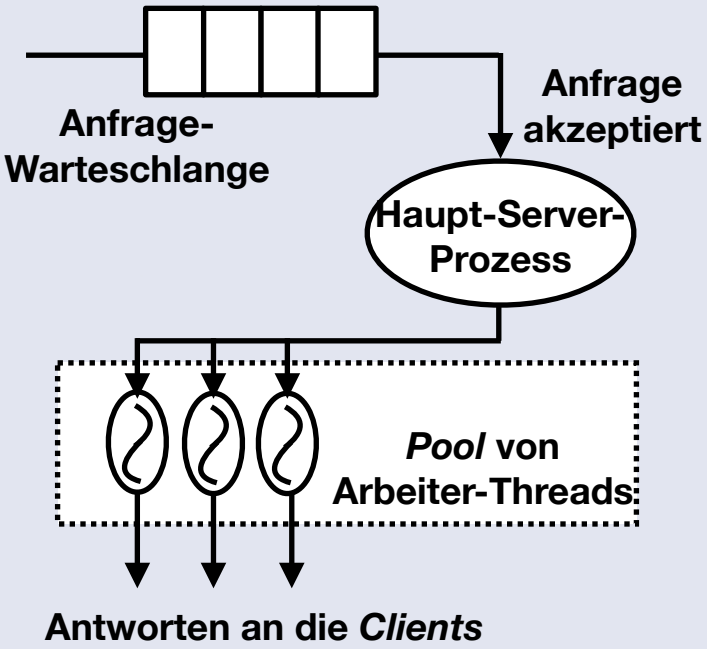
- Moderne Lösung: ***copy on write***
  - `fork()` mit *copy on write* ist *fast* so schnell wie `vfork()`
- Das **Gewicht** eines Prozesses ist eine *informelle* Beschreibung der Größe seines Kontextes
- Damit auch ein Indikator für die Zeit, die eine Kontextumschaltung benötigt
    - Diese umfasst (unter anderem):
      - CPU-Scheduling
      - Sichern des vorherigen Kontextes
      - Laden des neuen Kontextes
- Klassische Unix-Prozesse sind “schwergewichtig”
  - …egal, ob wir *copy on write* verwenden oder nicht

Leichtgewichtige Prozesse (Threads)

- Prozesse besitzen ein 1:1 Verhältnis zwischen **Kontrollfluss** und **Adressraum**
  - auch für mit *fork* und *copy on write* erzeugte Prozesse
- Kooperierende **Threads** (oder Fäden) **teilen** sich **einen Adressraum**
  - code + data + bss + heap-Segmente, aber **nicht** den Stack!
  - Warum nicht den Stack?
    - Jeder Thread hat einen eigenen Kontrollfluss
    - Benötigt eigene Aufrufhierarchie, lokale Variablen usw.
- Vorteil von Threads:
  - Komplexe Aufgaben können in einen leichtgewichtigen Hilfs-Thread ausgelagert werden
  - Der Eltern-Thread kann z.B. schon wieder auf Eingaben warten, während der Hilfs-Thread läuft ➛ reduzierte **Latenz** (Antwortzeit)

Beispiel für Threads

- Typische Anwendung für Threads: **Webserver**
- Programme mit unabhängigen Kontrollflüssen können direkt von Multiprozessorsystemen profitieren
- Schnelle Kontextumschaltung: kein Kopieren des Adressraums
  - nur den *stack pointer* umschalten – ein CPU-Register
- Nachteile von Threads:
  - Schwierige und fehleranfällige Programmierung
  - Zugriff auf gemeinsame Daten von Threads erfordert Koordination
  - Betriebssystem muss Threads verwalten ➛ Aufwand



nach [3]

Threads in Linux

- Linux implementiert **POSIX-Threads** in der **pthreads**-Library [4]
- pthreads unter Linux verwenden Linux-spezifischen Systemaufruf:

Linux Systemaufruf:

```
int __clone(int (*fn)(void*), void *stack, int flags,
void *arg)
```

- Universelle Funktion, parametrisiert durch den **flags**-Parameter:
  - `CLONE_VM` Verwende gemeinsamen Adressraum
  - `CLONE_FS` Teile Information über das Dateisystem
  - `CLONE_FILES` Teile Dateideskriptoren (geöffnete Dateien)

- Linux verwaltet intern alle **Threads** und **Prozesse** als ***Tasks***
  - Der Scheduler unterscheidet nicht zwischen diesen

Fibers

- *auch **Benutzer-level-Threads** oder **federgewichtige Prozesse** genannt*
- Implementiert auf Anwendungsebene (innerhalb eines Prozesses)
  - Das Betriebsssystem kennt *Fibers* nicht
  - Entsprechend werden immer **ganze Prozesse gescheduled**
- Implementiert über eine Library: *user level thread package*
- Vorteile:
  - Extrem schnelle Kontextumschaltung: nur die Prozessorregister austauschen!
  - Kein Umschalten in Kernelmodus zur Fiber-Umschalten notwendig
  - Jede Anwendung kann die bestgeeignete Fiber-Library wählen
- Nachteile:
  - Blockieren einer einzelnen Fiber blockiert den gesamten Betriebssystemprozess (da das OS die Fiber nicht kennt)
  - Geschwindigkeitsvorteil auf Multiprozessorsystemen aufwändig zu erhalten

Prozesse ↔ Threads ↔ Fibers

	Prozesse	Threads	Fibers
Adressraum	separat	geteilt	geteilt
Sichtbarkeit im Kernel	ja	ja	nein
Scheduling	Kernel	Kernel	Benutzer-modus
Stack	separat pro Prozess	separat pro Thread	*kann* geteilt werden
Umschalt-aufwand	sehr hoch	hoch	gering

CPU-Zuteilung (Scheduling)

- Zuteilung (*scheduling*) ermöglicht die Koordination nebenläufiger Prozesse (und Kernel-Threads)

- Grundsätzliche Fragen:
  - Welche Ereignisse sollten zu Verdrängung führen?
  - In welcher Reihenfolge sollten Prozesse ausgeführt werden?

- Optimierungsziele eines ***Zuteilungsalgorithmus***:
  - Benutzerorientiert ➛ kurze Reaktionszeiten
  - Systemorientiert ➛ optimale CPU-Ausnutzung

- Kein einzelner Zuteilungsalgorithmus erfüllt alle Anforderungen!

Prozesszustände (nochmal)

- **ready**: bereit, von der CPU ausgeführt zu werden
  - ein Prozess ist in der *ready (Warte-)Liste* für die CPU-Zuteilung
  - die Listenposition ist abhängig vom Zuteilungsalgorithmus
- **running**: die Ressource "CPU" ist dem Prozess zugeteilt
  - ein Prozess rechnet: *CPU burst*
  - es gibt *nur einen laufenden Prozess pro CPU* zu jedem gegebenen Zeitpunkt
- **blocked**: wartet auf ein Ereignis
  - ein Prozess führt Ein- oder Ausgabe durch: *I/O burst*
  - wartet auf das Eintreten mindestens einer Bedingung

Zustandsübergangspunkte

- Jeder Übergang zum **ready**-Zustand aktualisiert die **Warteschlange** der CPU
  - eine Entscheidung über die Einreihung des Prozesses erfolgt
  - das Ergebnis hängt von der ***CPU-Zuteilungsstrategie*** des Systems ab
- ***Zuteilung*** und ***Wiederzuteilung*** werden ausgeführt…
  1. nachdem ein neuer Prozess erzeugt wurde
  2. wenn ein Prozess die CPU abgibt
  3. wenn ein Ereignis, auf das ein Prozess wartet, eintritt

- Ein Prozess kann ***gezwungen werden***, die CPU abzugeben:
  → ***präemptive Zuteilung***, z.B. durch einen Zeitgeber-Interrupt

Zuteilungsstrategie: First-Come First-Served – FCFS

- Ein einfacher und fairer (?) Algorithmus: "first come first served"
- Reihenfolgekriterium ist die *Ankunftszeit* eines Prozesses
- Algorithmus ist *nicht präemptiv* und erwartet *kooperierende Prozesse*

Prozess	Zeiten					
	Ankunft	Benötigte Ausführungszeit T_s	Start	Ende	Laufzeit T_r	T_r/T_s
A	0	1	0	1	1	1,00
B	1	100	1	101	100	1,00
C	2	1	101	102	100	**100,00**
D	3	100	102	202	199	1,99
Durchschnitt						26,00

- Beispiel:
  - Die normalisierte Laufzeit (T_r / T_s) von C ist hoch im Vergleich zu seiner benötigten Ausführungszeit (*service time*) T_s

Diskussion: FCFS – "Konvoi-Effekt"

- Dieses Problem betrifft kurz laufende, **I/O-intensive Prozesse**, die auf lang laufende, **CPU-intensive Prozesse** folgen
  - Prozesse mit **langen** *CPU bursts* profitieren davon
  - Prozesse mit **kurzen** *CPU bursts* sind benachteiligt
- FCFS minimiert die Anzahl von *Kontextumschaltungen*
- Dieser **Konvoi-Effekt** erzeugt aber eine Reihe von Problemen:
  - Lange Antwortzeiten
  - Niedriger I/O-Durchsatz
- Wenn das System einen Mix von CPU- und I/O-intensiven Prozessen ausführt, ist FCFS kein brauchbarer Ansatz
  - Typischerweise nur in Systemen mit Stapelverarbeitung (*batch processing)* verwendet

Round Robin (RR)

- Reduziert die Benachteiligung von Prozessen mit kurzen *CPU bursts*:
  - die verfügbare Prozessorzeit wird in ***Zeitscheiben*** aufgeteilt
- Wenn eine Zeitscheibe aufgebraucht ist, ***kann*** eine Prozessumschaltung stattfinden
  - der unterbrochene Prozess wird ans Ende der *ready list* gesetzt
  - der nächste Prozess wird nach FCFS aus der *ready list* ausgewählt
- Basis für garantierten Zugriff auf die CPU:
  Zeitgeber erzwingt einen Interrupt am Ende jeder Zeitscheibe
- Die Effizienz dieses Ansatzes hängt insbesondere von der gewählten Länge der Zeitscheibe ab:
  - zu lang ➛ round robin degeneriert zu FCFS
  - zu kurz ➛ sehr hoher *overhead* für häufige Prozessumschaltungen

Diskussion: RR – Performanceprobleme

- I/O-intensive Prozesse beenden ihren *CPU burst* vor dem Ende ihrer Zeitscheibe
  - sie blockieren und werden wieder ans Ende der *ready list* gesetzt, wenn ihr *I/O burst* beendet ist
- CPU-intensive Prozesse dagegen nutzen ihre Zeitscheibe vollständig
  - sie werden dann verdrängt und direkt wieder ans Ende der *ready list* gesetzt
- Die Menge an CPU-Zeit, die Prozesse erhalten, ist daher ungleich verteilt ➛ CPU-intensive Prozesse erhalten größeren Anteil
  - I/O-intensive Prozesse werden nicht so gut bedient, daher ist die Ausnutzung von I/O-Geräten niedrig
  - Die *Varianz der Antwortzeit* von I/O-intensiven Prozessen steigt

Shortest Process Next (SPN)

- Reduziert die Benachteiligung kurzer *CPU bursts* bei FCFS: "der kürzeste kommt zuerst dran…"
  - dies erfordert Wissen über die Prozesslaufzeiten
  - keine Präemption
- Das Hauptproblem ist die ***Vorhersage der Laufzeiten***
  - Batchverarbeitung:
    der Programmierer annotiert die benötigte *Zeitschranke*
  - Interaktive Verarbeitung:
    Zeitschranke wird abgeschätzt basiert auf vorherigen *CPU burst*-Längen des Prozesses
- Antwortzeiten werden deutlich reduziert und die Gesamtsystemleistung verbessert
  - Aber: Gefahr des ***Verhungerns*** CPU-intensiver Prozesse

Diskussion: SPN – Gewichtung von bursts

- *CPU bursts*, die weiter in der Vergangenheit liegen, sollten geringer gewichtet werden. Die Vorhersage für die Dauer des nächsten CPU-Bursts τ_{n+1} ist:

$$\tau_{n+1} = \alpha \cdot T_n + (1 - \alpha) \cdot \tau_{n-1}$$

  - Werte des konstanten Gewichtungsfaktors α: $0 < \alpha < 1$
  - gibt die *relative Gewichtung* einzelner *CPU bursts* in der Zeitlinie des Prozesses an

Dieser ***statistische Ansatz*** heißt auch **exponentielle Glättung**

- Rekursives Lösen führt zu:

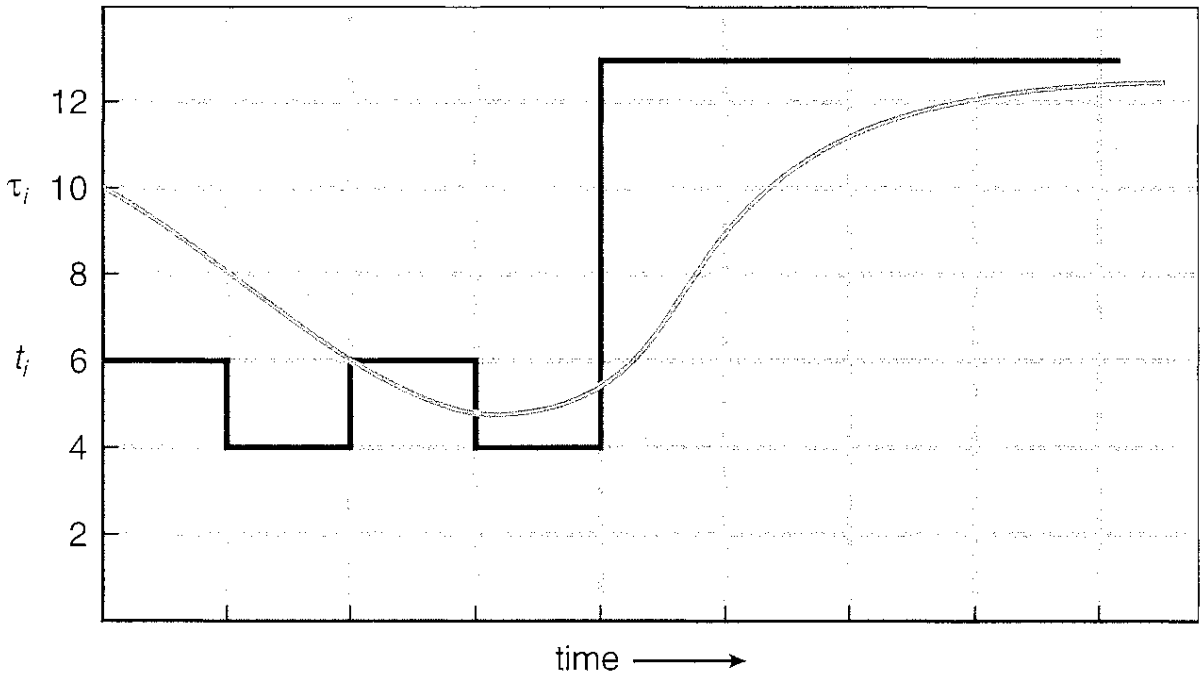
$$\tau_{n+1} = \alpha \cdot T_n + (1 - \alpha)\alpha T_{n-1} + \cdots + (1 - \alpha)\alpha T_{n-i} + \cdots + (1 - \alpha)^n \tau_1$$

$$\tau_{n+1} = \alpha \cdot \sum_{i=0}^{n-1} (1 - \alpha)^i T_{n-i} + (1 - \alpha)^n \tau_1$$

- für α = 0.8:

$$\tau_{n+1} = 0{,}8T_n + 0{,}16T_{n-1} + 0{,}032T_{n-2} + 0{,}0064T_{n-3} + \cdots$$

Diskussion:
SPN – Gewichtung von bursts



CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Vorhersage der Zeitdauer des nächsten CPU-Bursts (nach Siberschatz)

Ein weites Feld…

- Es gibt viele weitere Zuteilungsverfahren für Prozesse und Threads…
  - Besonders relevant im Bereich der *Echtzeitsysteme*
  - Hier nicht weiter betrachtet!

- Bei Interesse: mehr Details in [1] Kapitel 7–9

- **Problem** von Multitasking-Systemen:
  - Parallel ablaufende Prozesse müssen voneinander *isoliert* werden, damit die Sicherheit und Vertraulichkeit von Daten des Prozesses gewährleistet wird ➛ nächste Vorlesung…

Prozesse, nochmal…

- Prozesse sind **Programme in Ausführung** (unter Kontrolle des OS)
  - ***Die*** Abstraktion für Kontrollflüsse
  - Prozesse sind konzeptionell unabhängig
  - Die CPU wird ***multiplext***
  - Das OS bestimmt, wann ein Prozess verdrängt wird und in welcher Reihenfolge Prozesse ausgeführt werden
- Prozesse haben einen **Adressraum**
  - Logische Adressen eines Prozesses werden auf physische Adressen mit Hilfe der Hardware (MMU) abgebildet → nächste Woche
- Prozesse ***können*** Code- und Datenbereiche teilen
  - Threads und Fibers arbeiten ***immer*** auf dem selben Adressraum
  - Das OS kann einen einzelnen Speicherbereich über die MMU in mehrere Adressräume einblenden
  - Daten des OS selbst werden auch (kontrolliert) geteilt

Beispiel: Threads

Ein kleines Programm, das zwei Threads in C erzeugt:

```
void *mythread(void *arg) {
  printf("%s\n", (char *) arg); // übergebenes Argument drucken
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p1, p2;

  printf("main beginnt\n");
  pthread_create(&p1, NULL, mythread, "A"); // wir ignorieren hier
  pthread_create(&p2, NULL, mythread, "B"); // eventuelle Fehler!

  // join wartet auf Beendigung des jeweiligen Threads
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main endet\n");
  return 0;
}
```

Die pthread-Library ist Standard in Unix/Linux
Mit pthread_create wird ein neuer Thread erzeugt, der mit der angegebenen Funktion startet, mit pthread_join wird auf das Ende des angegebenen Threads gewartet

Ausgabe des Beispielcodes (1)

Welche Ausgabe wird erzeugt?

```
void *mythread(void *arg) {
  printf("%s\n", (char *) arg);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p1, p2;

  printf("main beginnt\n");
  pthread_create(&p1, NULL,
        mythread, "A");
  pthread_create(&p2, NULL,
        mythread, "B");

  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main endet\n");
  return 0;
}
```

main	Thread 1	Thread 2
startet		
druckt "main beginnt"		
erzeugt Thread 1		
erzeugt Thread 2		
wartet auf Thread 1		
	läuft	
	druckt "A"	
	return	
wartet auf Thread 2		
		läuft
		druckt "B"
		return
druckt "main endet"		

Ausgabe des Beispielcodes (2)

Welche Ausgabe wird erzeugt?

```
void *mythread(void *arg) {
  printf("%s\n", (char *) arg);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p1, p2;

  printf("main beginnt\n");
  pthread_create(&p1, NULL,
        mythread, "A");
  pthread_create(&p2, NULL,
        mythread, "B");

  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main endet\n");
  return 0;
}
```

main	Thread 1	Thread 2
startet		
druckt "main beginnt"		
erzeugt Thread 1		
	läuft	
	druckt "A"	
	return	
erzeugt Thread 2		
		läuft
		druckt "B"
		return
wartet auf Thread 1		
wartet auf Thread 2		
druckt "main endet"		

Schon beendet – kehrt sofort zurück!

Ausgabe des Beispielcodes (3)

Welche Ausgabe wird erzeugt?

```
void *mythread(void *arg) {
  printf("%s\n", (char *) arg);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p1, p2;

  printf("main beginnt\n");
  pthread_create(&p1, NULL,
        mythread, "A");
  pthread_create(&p2, NULL,
        mythread, "B");

  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main endet\n");
  return 0;
}
```

main	Thread 1	Thread 2
startet		
druckt "main beginnt"		
erzeugt Thread 1		
erzeugt Thread 2		
		läuft
		druckt "B"
		return
wartet auf Thread 1		
	läuft	
	druckt "A"	
	return	
wartet auf Thread 2		
druckt "main endet"		

Schon beendet – kehrt sofort zurück!

Probleme mit Threads

- Die **Reihenfolge** der Abarbeitung von Threads (wie bei Prozessen, die mit fork erzeugt worden) ist nicht vorhersagbar!
- Damit hängt das Ergebnis eines Programms, das Threads verwendet, von der Scheduling-Reihenfolge ab!

main	Thread 1	Thread 2
startet	**1: "AB"**	
druckt "main beginnt"		
erzeugt Thread 1		
erzeugt Thread 2		
wartet auf Thread 1		
	läuft	
	druckt "A"	
	return	
wartet auf Thread 2		
		läuft
		druckt "B"
		return
druckt "main endet"		

main	Thread 1	Thread 2
startet	**2: "AB"**	
druckt "main beginnt"		
erzeugt Thread 1		
	läuft	
	druckt "A"	
	return	
erzeugt Thread 2		
		läuft
		druckt "B"
		return
wartet auf Thread 1		
wartet auf Thread 2		
druckt "main endet"		

main	Thread 1	Thread 2
startet	**3: "BA"**	
druckt "main beginnt"		
erzeugt Thread 1		
erzeugt Thread 2		
		läuft
		druckt "B"
		return
wartet auf Thread 1		
	läuft	
	druckt "A"	
	return	
wartet auf Thread 2		
druckt "main endet"		

Beispiel: Gemeinsame Daten

Ein Programm, das einen Zähler in Threads hochzählt:

```
volatile int counter = 0;

void *mythread(void *arg) {
  for (int i = 0; i < 10000000; i++) {
    counter = counter + 1;
  }
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p1, p2;

  printf("main startet, counter = %d\n", counter);
  pthread_create(&p1, NULL, mythread, 0);
  pthread_create(&p2, NULL, mythread, 0);
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main endet, counter = %d\n", counter);
  return 0;
}
```

10 Millionen!

Erwartetes Ergebnis

```
$ ./counter
main startet, counter = 0
main endet, counter = 20000000
```

Gemeinsame Daten – Ausgabe

Universität Bamberg

```c
volatile int counter = 0;

void *mythread(void *arg) {
  for (int i = 0; i < 10000000; i++) {
    counter = counter + 1;
  }
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p1, p2;

  printf("main startet, counter = %d\n",
    counter);
  pthread_create(&p1, NULL, mythread, 0);
  pthread_create(&p2, NULL, mythread, 0);
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main endet, counter = %d\n",
    counter);
  return 0;
}
```

```
$ ./counter
main startet, counter = 0
main endet, counter = 19576684
```

```
$ ./counter
main startet, counter = 0
main endet, counter = 19345221
```

```
$ ./counter
main startet, counter = 0
main endet, counter = 19221041
```

Gemeinsame Daten – Problem

Universität Bamberg

```
volatile int counter = 0;

void *mythread(void *arg) {
  for (int i = 0; i < 10000000; i++) {
    counter = counter + 1;
  }
  return NULL;
}
```

```
riscv64-unknown-elf-gcc -S -O2 count.c
```

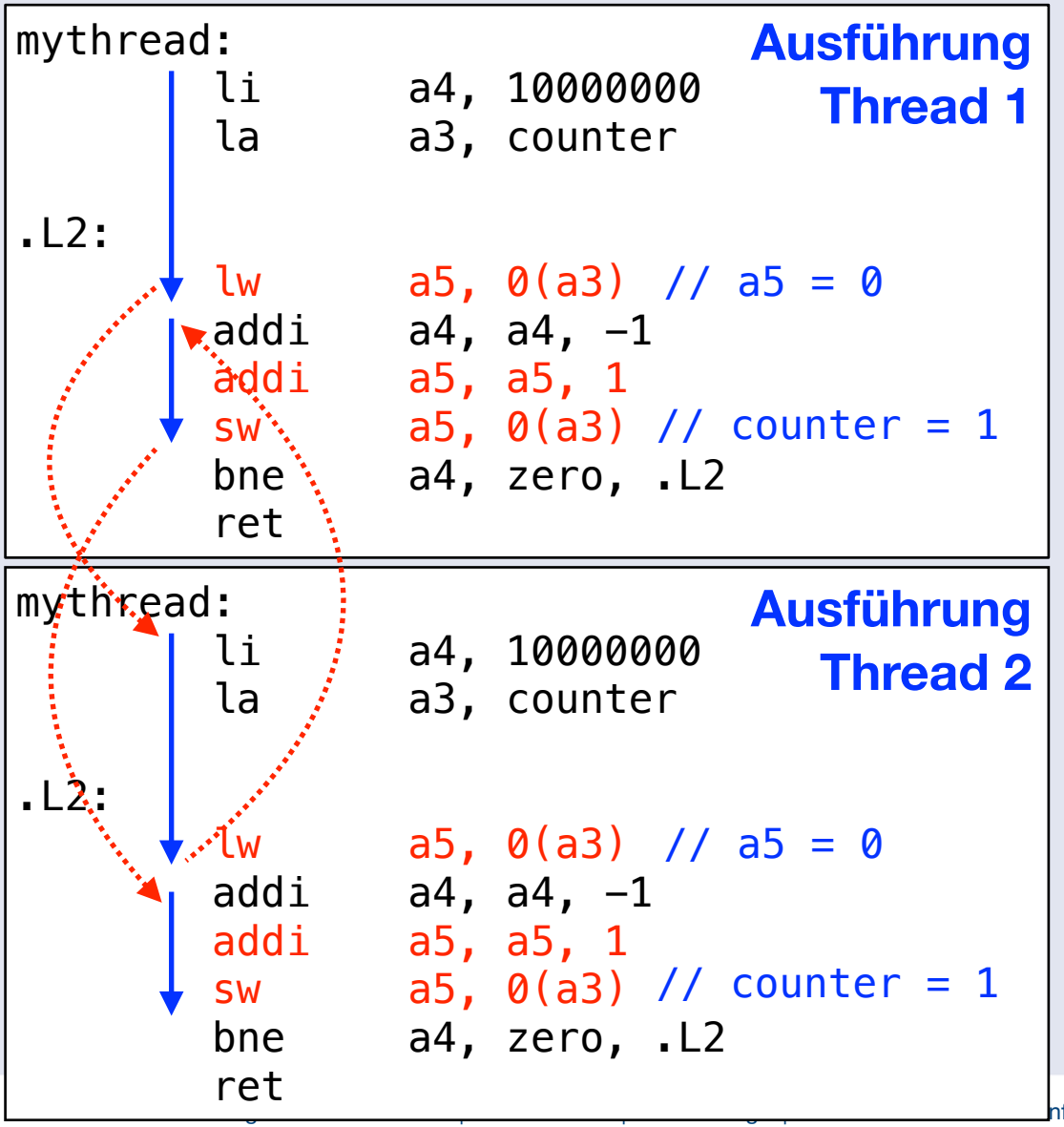
```
mythread:
        li      a4, 10000000
        la      a3, counter
.L2:
        lw      a5, 0(a3)
        addi    a4, a4, -1
        addi    a5, a5, 1
        sw      a5, 0(a3)
        bne     a4, zero, .L2
        ret
```

Beim Hochzählen von `counter` gehen Werte verloren – warum?

Der einzelne C-Befehl counter = counter+1 wird vom Compiler in drei Maschinenbefehle übersetzt!

Gemeinsame Daten – Analyse

Universität Bamberg



Bei der Ausführung beider Threads erfolgt ab und zu eine Prozessumschaltung **zwischen** Laden und Rückschreiben des Zählerwertes

Als Folge wird ab und zu **zweimal der selbe Ausgangswert** geladen, inkrementiert und zurückgeschrieben!

Das Problem: *race conditions*

Universität Bamberg

- Eine ***race condition*** (Wettlaufsituation) ist eine Situation, bei der mehrere Threads *nebenläufig auf gemeinsame Daten zugreifen* und mindestens einer der Prozesse die Daten *verändert*
  - Wenn eine *race condition* eintritt, hängt der resultierende Wert der gemeinsamen Daten von der Reihenfolge ab, in der die Threads darauf zugreifen
  - Wir bezeichnen die Instruktionsfolge, die auf die gemeinsamen Daten zugreift, als ***kritischen Abschnitt***
  - Das Ergebnis ist daher ***nicht vorhersagbar*** und kann bei überlappenden Threads sogar ***inkorrekt*** sein!

- Zur Vermeidung von *race conditions* müssen wir nebenläufige Threads ***synchronisieren***

Wo tritt dieses Problem auf?

- **Geteilter Speicher**, der zur Kommunikation zwischen Prozessen verwendet wird
  - Bei Systemen, die einen solchen Service anbieten
- **Threads und Fibers**
  - Konkurrierender Zugriff auf die selben Variablen
- **Betriebssystemdaten**, mit denen der Zugriff auf nicht-teilbare Ressourcen koordiniert wird
  - Dateisystemstrukturen, Prozesstabelle, Speicherverwaltg., …
  - Geräte (Terminals, Drucker, Netzwerkschnittstellen, …)
- Ähnlicher Spezialfall: **Interruptsynchronisation**
  - Achtung: Methoden zur Synchronisation von Prozessen funktionieren nicht notwendigerweise auch für Interrupts!

Def.: Synchronisation

- Die Koordination der Kooperation und Konkurrenz zwischen Prozessen wird ***Synchronisation*** genannt
  - Eine Synchronisation bringt die Aktivitäten verschiedener nebenläufiger Prozesse in eine **Reihenfolge**
  - Durch sie erreicht man also prozessübergreifend das, wofür innerhalb eines Prozesses die Sequentialität von Aktivitäten sorgt

Quelle: Herrtwich/Hommel (1989), Kooperation und Konkurrenz, p. 26

Def.: Kritischer Abschnitt

- Im Falle einer ***race condition*** konkurrieren N Prozesse um den Zugriff auf gemeinsame Daten

- Die Instruktionsfolgen (Codefragmente), die auf diese kritischen Daten zugreifen, nennt man ***kritische Abschnitte*** *(critical sections)*

- **Problem**
  - Wir müssen sicherstellen, dass nur ein einziger Prozess gleichzeitig den Code in einem kritischen Abschnitt ausführt

Lösung: Lockvariablen

Eine Lockvariable (Schloßvariable) ist ein abstrakter Datentyp mit zwei Operationen: `acquire` und `release`

```
lock_t mutex; // global allozierter Lock 'mutex'
...
acquire(&mutex);

counter = counter + 1;

release(&mutex);
```

- **blockiert den Prozess, bis das angegebene Lock offen ist**
- **schließt das Lock dann selbst "von innen" ab**

- **öffnet das angegebene Lock, ohne den aufrufenden Prozess zu blockieren**

Implementierung von Locks: inkorrekt

Diese naive Lock-Implementierung funktioniert nicht!

```
// Lockvariable (Initialwert ist 0)
typedef unsigned char Lock;

// Kritischen Abschnitt betreten
void acquire (Lock *lock) {
  while (*lock); /* Achtung: leerer Schleifenkörper! */
  *lock = 1;
}

// Kritischen Abschnitt verlassen
void release (Lock *lock) {
  *lock = 0;
}
```

Implementierung von Locks: inkorrekt


```
// Lockvariable
typedef unsigned char Lock;

// Kritischen Abschn betreten
void acquire (Lock *lock) {
  while (*lock);
  *lock = 1;
}

// Kritischen Abschn. verlassen
void release (Lock *lock) {
  *lock = 0;
}
```

`acquire` muss den kritischen Abschnitt schützen – ***ist aber selbst kritisch!***

- Der kritische Augenblick ist der Zeitpunkt nach Verlassen der Warteschleife und vor Setzen der Lockvariable!
- Wenn der aktuelle Prozess **zwischen der Ausführung** der beiden Codezeilen verdrängt wird, sieht ein anderer Prozess den kritischen Abschnitt als frei an und betritt ihn ebenfalls!

Wenn dies geschieht, können (mindestens) zwei Prozesse *gleichzeitig* den kritischen Abschnitt betreten, der von `acquire` geschützt werden sollte!

Unterdrücken von Interrupts

Was ist der Grund für eine Prozessumschaltung in einem kritischen Abschnitt?

- Das Betriebssystem greift ein und setzt einen anderen Prozess in den Zustand RUNNING
- Dies kann nur geschehen, wenn das **OS die Kontrolle übernimmt**
  ➛ ein Timer- oder Geräte-**Interrupt** tritt auf

Idee: **Interrupts abschalten, um sicherzustellen, dass ein Prozess in seinem kritischen Abschnitt nicht unterbrochen wird!**

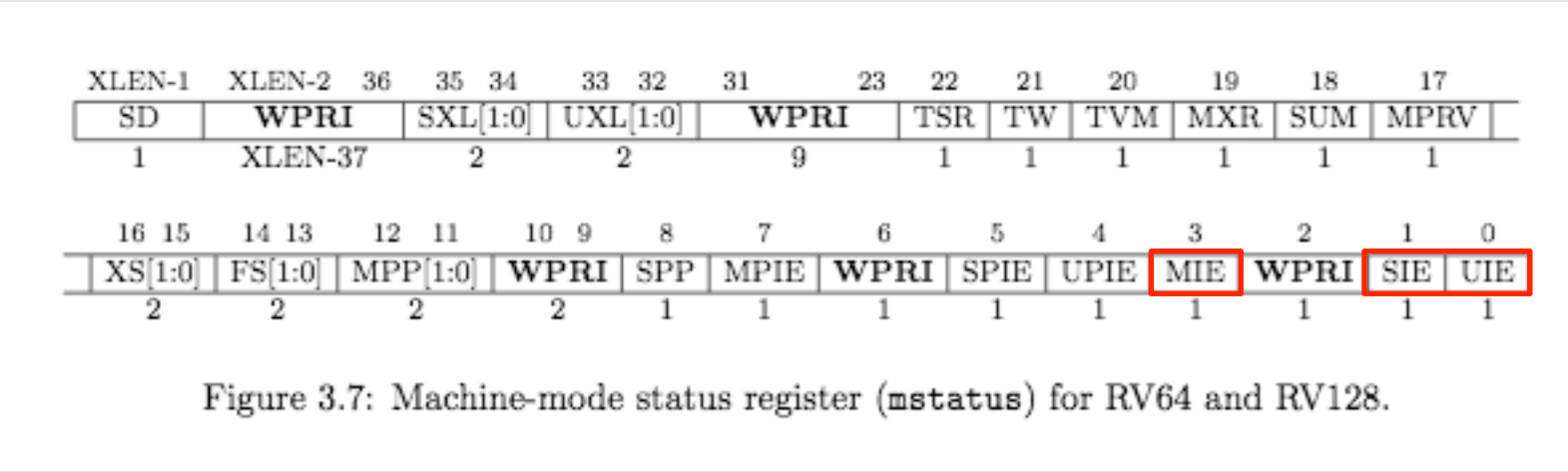
```
// Krit. Absch. betreten
void acquire (Lock *lock) {
  asm ("cli");
}

// Krit. Absch. verlassen
void release (Lock *lock) {
  asm ("sti");
}
```

`cli` und `sti` werden bei x86-Prozessoren verwendet, um Interrupts ab- und anzuschalten

Interrupts bei RISC-V steuern

Universität Bamberg

Bei RISC-V wird das Abschalten von Interrupts durch **Enable-Bits** in einem spezielles Register (csr, *configuration and status register)* gesteuert:



XLEN-1	XLEN-2 36	35 34	33 32	31 23	22	21	20	19	18	17	
SD	**WPRI**	SXL[1:0]	UXL[1:0]	**WPRI**	TSR	TW	TVM	MXR	SUM	MPRV	
1	XLEN-37	2	2	9	1	1	1	1	1	1	

16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
XS[1:0]	FS[1:0]	MPP[1:0]	**WPRI**	SPP	MPIE	**WPRI**	SPIE	UPIE	MIE	**WPRI**	SIE	UIE
2	2	2	2	1	1	1	1	1	1	1	1	1

Figure 3.7: Machine-mode status register (`mstatus`) for RV64 and RV128.

Der zugehörige Code ist etwas aufwendiger…
bei Interesse im xv6-Code nachschauen:
https://github.com/mit-pdos/xv6-riscv/blob/riscv/kernel/start.c

Warum ist das Ab-/Abschalten von Interrupts keine gute Idee?

Locks mit atomaren Instruktionen

Viele Prozessoren unterstützten unteilbare (**atomare**) Lese/Ändern/Schreibzyklen, die zur Implementierung von Lock-Algorithmen verwendet werden können

- Spezielle Maschineninstruktionen für atomare Operationen:
  - Motorola 68K: TAS (test and set)
    - setzt Bit 7 des Zieloperanden und gibt vorherigen Zustand in den Status-flags in der CPU zurück
  - Intel x86: XCHG (exchange)
    - Vertauscht den Inhalt eines Registers mit dem einer Speicherzelle (also einer Variablen im Speicher)
  - RISC-V: AMOxxx (atomic memory operation)
    - Unterstützte Operationen xxx:
      - SWAP, ADD, AND, OR, XOR, MAX, MIN

```
acquire  TAS lock
         BNE acquire
```

```
         mov  ax, 1
acquire  xchg lock
         cmp  ax, 0
         jne  acquire
```

```
v1 = *addr
*addr = OP(v1, v2)
```

Diskussion: Aktives Warten

- Unsere bisherigen Lockalgorithmen haben einen entscheidenden Nachteil: **Aktives Warten**

  Der **aktiv wartende** Prozess…
  - kann nicht selbst die Bedingung beeinflussen, auf die er wartet
  - behindert unnötigerweise andere Prozesses, die die CPU für "nützliche" Arbeit verwenden können
  - behindert sich selbst durch aktives Warten:
    - Je länger ein Prozess den Prozessor hält, desto länger muss er warten, bis andere Prozesse seine Wartebedingung erfüllt haben
    - Dieses Problem ist spezifisch für Einprozessorsysteme

Alternative: *passives Warten*

- Idee: Prozesse geben die CPU frei, während sie warten
  - Bei Synchronisation blockiert ein Prozess sich selbst, während er auf ein Ereignis wartet
    - Der Prozess wird in eine Warteschlange eingereiht
  - Wenn das Ereignis eintritt, dann wird ***einer der Prozesse***, die darauf warten, deblockiert (es kann mehr als einen geben)
- Die Wartephase eines Prozesses wird als Blockierungsphase realisiert (wie ein I/O burst)
  - Der Prozess-Scheduler wird aktualisiert
  - Ein anderer Prozess im Zustand READY wird auf RUNNING gesetzt (*dispatching*)
  - *Was geschieht, wenn zu dem Zeitpunkt kein Prozess READY ist?*
- Mit dem Beginn der Blockierungsphase eines Prozesses endet sein CPU-Burst

Referenzen

[1] Remzi Arpaci-Dusseau, Andrea Arpaci-Dusseau
Operating Systems: Three Easy Pieces
https://pages.cs.wisc.edu/~remzi/OSTEP/

[2] W. Stevens, Stephen Rago
Advanced Programming in the UNIX Environment
3rd Edition, Addison-Wesley 2013, ISBN-13: 978-0321637734

[3] Stavros Papastavrou et al.
Fine-Grained Parallelism in Dynamic Web Content Generation: The Parse and Dispatch Approach
Springer LNCS vol. 2888, 2003, doi 10.1007/978-3-540-39964-3_35

[4] David R. Butenhof
Programming with POSIX Threads
Addison-Wesley 1997, ISBN 978-0-201-63392-4