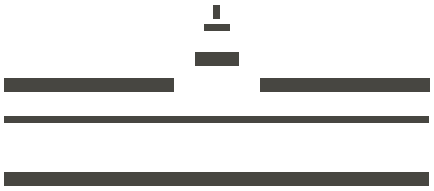


Westfälische
Wilhelms-Universität
Münster

> Übung Betriebssysteme

Übung 7

Wintersemester 2022/23

wissen.leben
WWU Münster

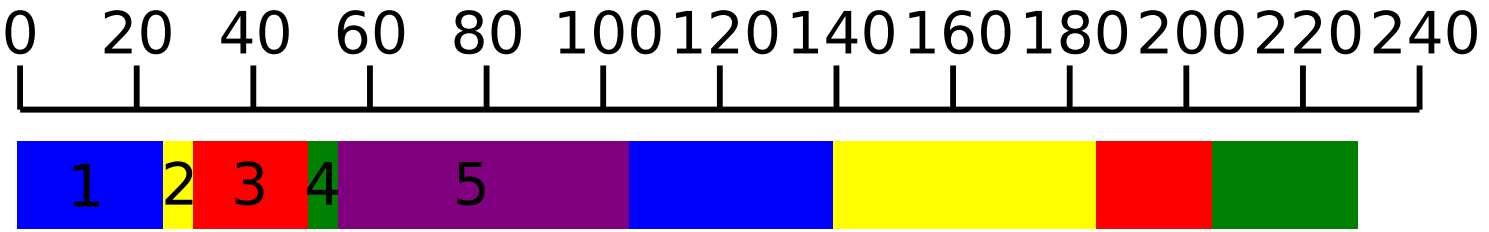
Gruppe Parallele und Verteilte Systeme (PVS)
Institut für Informatik
Westfälische Wilhelms-Universität Münster

Aufgabe 7
Scheduling-Strategien

Prozess	Ankunftszeit	Bedienzeit
P_1	0	60
P_2	25	50
P_3	30	40
P_4	50	30
P_5	55	50

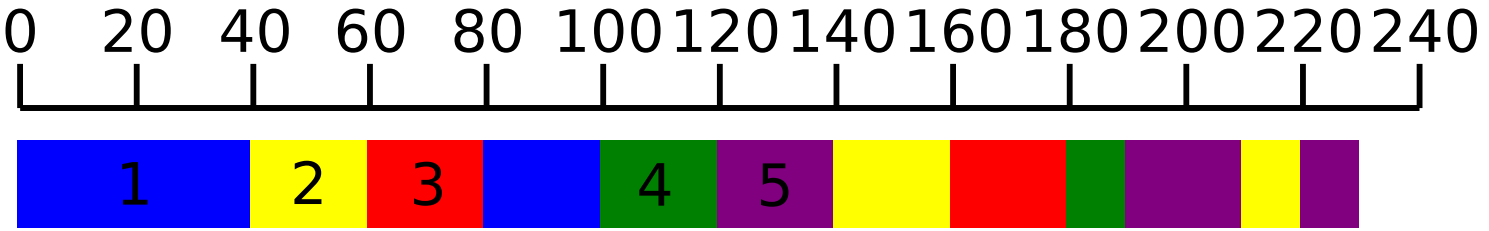
- Strategien: LCFS-PR, RR, Multilevel Feedback
- Gesucht: mittlere Bedien-, Warte- und Antwortzeit; normalisierte Antwortzeit für 1 und 4

Last Come First Served – Preemptive Resume



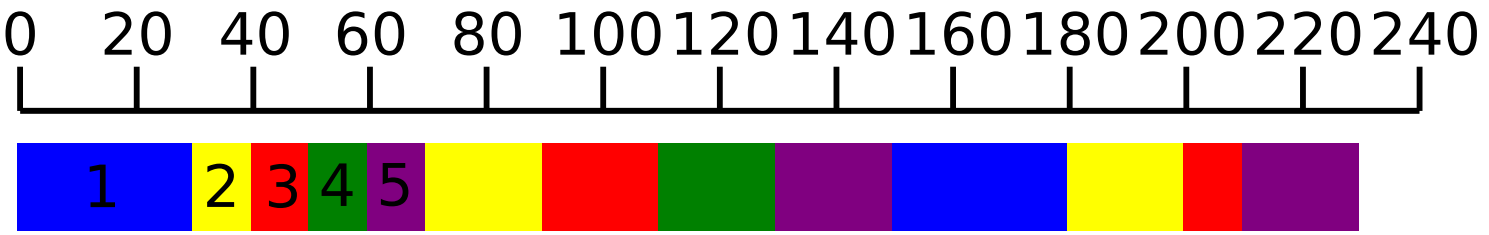
	P_1	P_2	P_3	P_4	P_5	ϕ
Ankunftszeit	0	25	30	50	55	
Bedienzeit	60	50	40	30	50	46
Startzeitpunkte	0	25	30	50	55	
	105	140	185	205		
Endzeitpunkt	140	185	205	230	105	
Wartezeit	80	110	135	150	0	95
Antwortzeit	140	160	175	180	50	141
Norm. Antwortzeit	$2\frac{1}{3}$	3,2	4,375	6	1	3,38

Round Robin (Zeitquantum 20)



	P_1	P_2	P_3	P_4	P_5	ϕ
Ankunftszeit	0	25	30	50	55	
Bedienzeit	60	50	40	30	50	46
Startzeitpunkte	0					
	20	40	60			
	80			100	120	
		140	160	180	190	
		210			220	
Endzeitpunkt	100	220	180	190	230	
Wartezeit	40	145	110	110	125	106
Antwortzeit	100	195	150	140	175	152
Norm. Antwortzeit	$1\frac{2}{3}$	3,9	3,75	$4\frac{2}{3}$	3,5	3,50

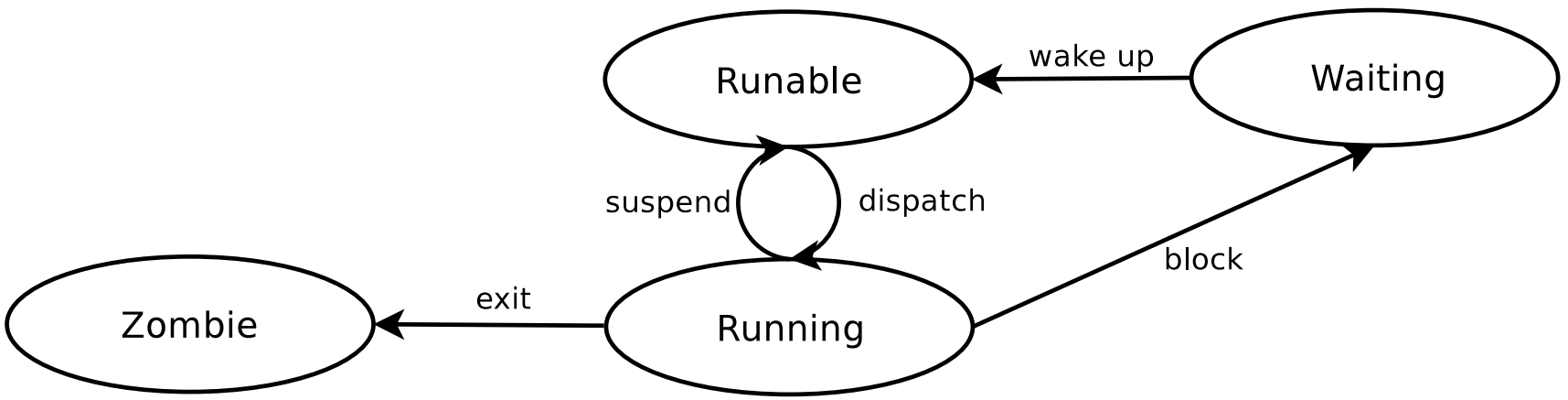
Multilevel Feedback (Zeitquantum $10 * (i + 1)$)



	P_1	P_2	P_3	P_4	P_5	ϕ
Ankunftszeit	0	25	30	50	55	
Bedienzeit	60	50	40	30	50	46
Startzeitpunkte	0					
	10	30	40	50	60	
		70	90	110	130	
	150	180	200		210	
Endzeitpunkt	180	200	210	130	230	
Wartezeit	120	125	140	50	125	112
Antwortzeit	180	175	180	80	175	158
Norm. Antwortzeit	3	3,5	4,5	$2\frac{2}{3}$	3,5	3,43

MiniOS: Ein kleines Betriebssystem

- Programm, das einfaches Betriebssystem simuliert
- Erlaubt „Blick unter die Haube“ eines BS
- Möglichkeit, u.a. verschiedene Schedulingstrategien zu implementieren
- Zustandsdiagramm eines Prozesses:



MiniOS: Übersicht

- Verwaltet mehrere Threads auf Benutzerebene („Prozesse“)
- Stellt Systemaufrufe, z. B. zur Erzeugung von Prozessen zur Verfügung
- Einfacher Scheduler schaltet zwischen Prozessen um

userspace		init.c proc1.c proc2.c mini_os.h	structs.h mini_os.h
mini_os	**queue/**	queue.h queue_pop.c queue_remove.c queue_search_id.c queue_push_bottom.c	ptable.c
	scheduler/	scheduler.h schedule.c start_scheduler.c select_next_process.c	globals.h constants.h
	syscalls/	syscalls.h create_process.c create_signal.c swait.c snotify.c	mini_os.c

Prozesszustände (constants.h):

```
enum process_state {
  STATE_CREATED    = 0,
  STATE_RUNNABLE   = 1,
  STATE_RUNNING    = 2,
  STATE_BLOCKED    = 3,
  STATE_DEAD       = 4,
};
```

Prozessstruktur (structs.h):

```
struct process {
  ucontext_t context;
  char name[32];
  struct process *next;
  struct process *prev;
  struct process *waiting_for_process;
  struct process *created_by;
  struct process_queue *in_queue;
  int (*process)(void);
  void *arg;
  int process_id;
  int exit_code;
  int backup_errno;
  enum process_state state;
};
```

Warteschlangen (structs.h)

```
struct process_queue {
  struct process *head;
  struct process *tail;
};

struct process_table {
  ucontext_t scheduler_context;
  struct process *running;
  struct process_queue run_queue;
  struct process_queue wait_queue;
  struct process_queue zombie_queue;
  int process_id;
};
```

- Beginn und Ende der WS in `process_queue` gekapselt; Verkettung der Prozesse über `process.next`
- Ein Prozess kann zu jedem Zeitpunkt in maximal einer WS sein
- Leere WS werden durch `first/last == NULL` gekennzeichnet

Warteschlangen II (queue/queue.h)

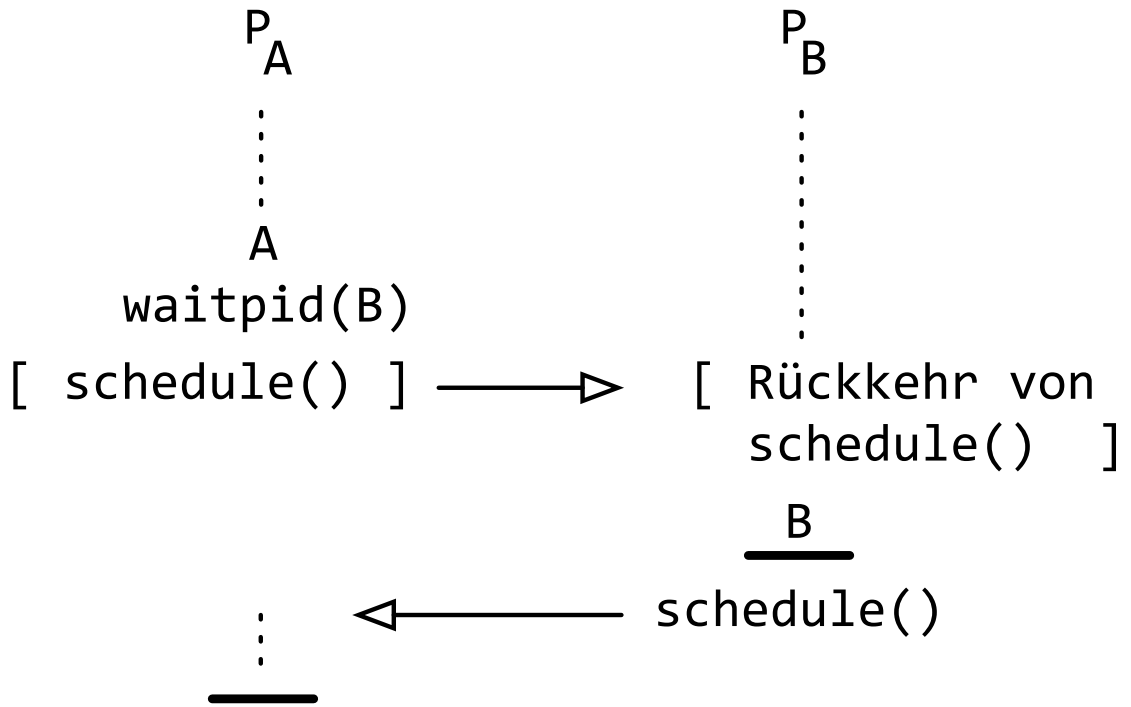
Warteschlange bearbeiten

```
struct process * queue_pop(struct process_queue *);
int queue_push_top(struct process_queue *, struct process *);
int queue_push_bottom(struct process_queue *, struct process *);
struct process * queue_search_id(struct process_queue *, int id);
int queue_swap(struct process *, struct process *);
int queue_remove(struct process *);
```

Operationen zum

- entfernen des ersten Prozesses aus einer `process_queue`
- hinzufügen eines neuen Prozesses am Anfang/Ende einer `process_queue`
- suchen eines Prozesses in einer `process_queue`
- vertauschen von zwei Prozessen
- entfernen eines Prozesses aus seiner `process_queue`

- `schedule`-Funktion wechselt zum nächsten Prozess in der `run_queue`
- Dazu kehrt die Funktion in den nächsten Prozess zurück
- Umgesetzt durch Kontextwechsel
- Beispiel: P_A: `A; wait_for_process(B);` P_B: `B;`
- Annahme: P_A wird ausgeführt



process_wrapper (scheduler/start_scheduler.c)

Wird aufgerufen, wenn ein Prozess erstellt wird

```
static void
process_wrapper(void)
{
  struct process *p = ptable.running;
  p->exit_code = p->process();
  p->state = STATE_DEAD;
  if (setcontext(&ptable.scheduler_context) == -1){
    perror("Failed to return to core scheduler");
    exit(EXIT_FAILURE);
  }

  // Should never return here
  fputs("Failed to return to core scheduler!\n", stderr);
  exit(EXIT_FAILURE);
```

Prozessumschaltung (`scheduler/{,start_}schedule.c`)

Modul zur Prozessumschaltung (verwendet `glibc` Funktionen)
Schnittstelle:

- Durch `schedule()` kann von außen ein Prozesswechsel angestoßen werden
- `start_scheduler()`: Startet eigentlichen Umschaltmechanismus
  - Speichert aktuellen Kontext als `ptable.scheduler_context` und lädt dann den Kontext des nächsten Prozess der von `select_next_process()` ausgewählt wurde
  - Wird das nächste Mal `schedule()` aufgerufen wird wieder der Kontext `ptable.scheduler_context` geladen und ein neuer Prozess ausgewählt und so weiter...

Vorgehensweise:

1. Bisher laufenden Prozess „aufräumen“, d. h. in die richtige Warteschlange einsortieren
2. Nächsten Prozess auswählen
3. Zum neu ausgewählten Prozess umschalten

Wegen der Prozessumschaltung kehrt Funktion `schedule` in einem anderen Prozess zurück, als sie aufgerufen wurde!

schedule I (scheduler/schedule.c)

```
void schedule(void)
{
  if (ptable.running->state == STATE_RUNNING)
    ptable.running->state = STATE_RUNNABLE;

  getcontext(&ptable.running->context);
  if (ptable.running->state == STATE_RUNNING){
    // Returned from scheduler ==> Restore errno and continue
    errno = ptable.running->backup_errno;
    return;
  }
  // Backup current errno value
  ptable.running->backup_errno = errno;
  // Return to core scheduler logic
  setcontext(&ptable.scheduler_context);
  // Should never return to this point
  fputs("Return to core scheduler failed!\n", stderr);
  exit(EXIT_FAILURE);
}
```

schedule II (scheduler/start_scheduler.c)

```
int start_scheduler(void)
{
  struct process *cur;
  if (getcontext(&ptable.scheduler_context) == -1){
    perror("Failed to back up scheduler context");
    return -1;
  }

  if (ptable.running != NULL){
    cur = ptable.running;
    if (cur->state == STATE_DEAD){
      // Resumed from process_wrapper() ==> Process ended
      if (handle_process_ended(cur))
        return -1;
    } else {
      // Resumed from schedule()
      if (handle_process_paused(ptable.running))
        return -1;
    }
  }
```

schedule main loop (scheduler/start_scheduler.c)

```
  while (ptable.run_queue.head){
    cur = ptable.running = select_next_process();
    if (cur->state == STATE_CREATED){
      // New process, start function
      cur->state = STATE_RUNNING;
      makecontext(&cur->context, process_wrapper, 0);
      setcontext(&cur->context);
      // Should not return to this point
    } else {
      // Existing process, continue function
      cur->state = STATE_RUNNING;
      setcontext(&cur->context);
      // Programm flow should never return here
    }
    if (handle_process_ended(cur))
      return -1;
  }
  return 0;
}
```

Scheduler Main-Loop

1. Initialisere `ptable.scheduler_context` mit dem aktuellen Kontext (`getcontext(&ptable.scheduler_context)`). Jeder Aufruf von `schedule()` setzt die Ausführung an dieser Stelle fort.
2. Prüfe, ob es einen momentan laufenden Prozess gibt. Dieser ist entweder terminiert oder wurde aus anderen Gründen angehalten.
3. Ausführung der Main-Loop bis es keine Prozesse mehr in `ptable.run_queue` mehr gibt.
   - Suche nächsten auszuführenden Prozess per `select_next_process()`
   - Falls der ausgesuchte Prozess das erste Mal läuft wird ein neuer Kontext für ihn erstellt
   - Es wird zu dem Kontext des ausgewählten Prozesses gewechselt

- Prozessauswahl nach größter PID

select_next_process (scheduler/select_next_process.c)

```
struct process *select_next_process(void)
{
  struct process *sel;

  sel = ptable.run_queue.head;

  for (struct process *cur = sel->next; cur ; cur = cur->next) {
    if (cur->process_id > sel->process_id) {
      sel = cur;
    }
  }

  queue_remove(sel);
  return sel;
}
```

main-Funktion (mini_os.c)

Initialisierung des ersten Prozesses und des Schedulers

```
int main(void)
{
  if (create_process("init", init, NULL) == -1){
    perror("Creating init process failed");
    return EXIT_FAILURE;
  }

  if (start_scheduler()){
    perror("Scheduling failed");
    return EXIT_FAILURE;
  }

  puts("Scheduler returned, all processes done");
  return EXIT_SUCCESS;
}
```

Systemaufruf: create_process (syscalls/create_process.c)

Erzeugung neuer Prozesse

```
#define STACK_SIZE 16384
int create_process(const char *name, int func(void), void *arg) {
  size_t name_len;
  struct process *result;
  result = malloc(sizeof(struct process));
  memcpy(result->name, name, name_len+1);
  result->next = NULL; result->prev = NULL;
  result->waiting_for_process = NULL;
  result->created_by = ptable.running;
  result->in_queue = NULL;
  result->process = func; result->arg = arg;
  result->process_id = ++ptable.process_id;
  result->state = STATE_CREATED;
  result->backup_errno = 0;
  getcontext(&result->context);
  result->context.uc_stack.ss_sp = malloc(STACK_SIZE);
  result->context.uc_stack.ss_size = STACK_SIZE;
  result->context.uc_link = &ptable.scheduler_context;
  queue_push_bottom(&ptable.run_queue, result);
  return result->process_id;
}
```

Systemaufruf: wait_for_process (syscalls/wait_for_process.c)

Auf Prozess warten

```
void wait_for_process(int pid)
{
  struct process *p;
  if ((p = queue_search_id(&ptable.run_queue, pid)) == NULL)
    p = queue_search_id(&ptable.wait_queue, pid);

  if (p == NULL){
    // Process with id "pid" no longer active, no need to wait
    return;
  }

  ptable.running->state = STATE_BLOCKED;
  ptable.running->waiting_for_process = p;
  printf("Process \"%s\" (PID %i) is waiting for process\
         \"%s\" (PID %i)\n",
         ptable.running->name, ptable.running->process_id,
         p->name, p->process_id);
  schedule();
```

Systemaufruf: msg (syscalls/msg.c)

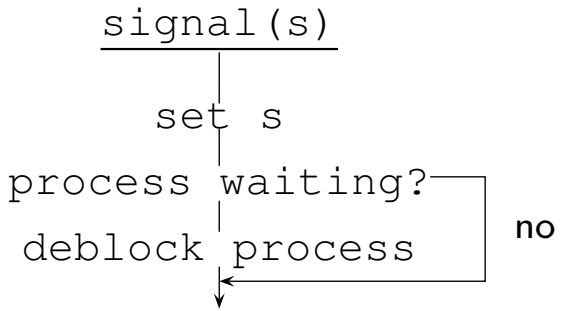
Ausgabe

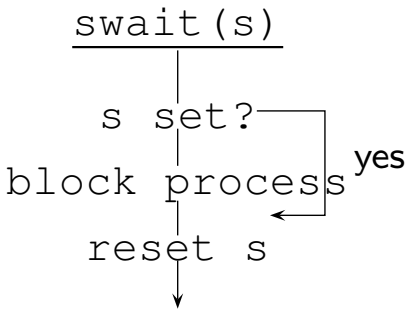
```
int msg(const char *s, ...)
{
  va_list ap;
  if (!s){
    errno = EINVAL;
    return -1;
  }

  printf("Message from process \"%s\" (PID %i):\n",
         ptable.running->name, ptable.running->process_id);
  va_start(ap, s);
  vprintf(s, ap);
  va_end(ap);
  puts("");
  return 0;
}
```

Systemaufruf: snotify/swait (syscalls/{snotify,swait}.c)

- Wie in der Vorlesung, Kapitel 4: (Signalisieren mit Wartezustand, nur ein Prozess wird aufgeweckt):





Datentypen, Definitionen (structs.h)

```
struct signal {
  struct process *waiting_process;
  int value;
};
```

Systemaufruf: snotify/swait (syscalls/create_signal.c)

Neue Signalvariablen bereitstellen

```
struct signal *create_signal(void)
{
  struct signal *result = malloc(sizeof(struct signal));

  if (result){
    result->waiting_process = NULL;
    result->value = 0;
  } else {
    errno = ENOMEM;
  }

  return result;
}
```

Systemaufruf: snotify/swait (syscalls/delete_signal.c)

Signalvariablen freigeben

```
int delete_signal(struct signal *s)
{
  if (s->waiting_process){
    fprintf(stderr, "Signal to delete still in use by\
            \"%s\" (PID %i)!\n",
            s->waiting_process->name,
            s->waiting_process->process_id);

    errno = EBUSY;
    return -1;
  }

  free(s);
  return 0;
}
```

Systemaufruf: snotify/swait (syscalls/swait.c)

swait

```
int swait(struct signal *s)
{
  if (s->waiting_process){
    errno = EBUSY;
    return -1;
  }

  if (s->value){
    // Signal already set :-) We can return immediately
    s->value = 0;
    return 0;
  }

  // Signal not set, we need to wait

  s->waiting_process = ptable.running;
  ptable.running->state = STATE_BLOCKED;
  schedule();
  return 0;
}
```

Systemaufruf: snotify/swait (syscalls/snotify.c)

snotify

```
int snotify(struct signal *s)
{
  if (s->waiting_process){
    // Notify waiting process
    s->waiting_process->state = STATE_RUNNABLE;
    if (
      (queue_remove(s->waiting_process) == -1) ||
      (queue_push_bottom(&ptable.run_queue, s->waiting_process))
    ){
      perror("Failed to move process from wait to run queue");
      exit(EXIT_FAILURE);
    }
    s->waiting_process = NULL;
  } else {
    // No process waiting, setting signal value to 1
    // so next swait() can return immediately
    s->value = 1;
  }

  return 0;
}
```

- Enthält „Benutzerprozesse“
- Jeder Prozess wird durch eine Funktion `int process(void)` implementiert
- Funktionen in `userspace` sollten ausschließlich auf Funktionen aus `syscalls/syscalls.h` zugreifen und keine Unix-Systemaufrufe tätigen
- Spezieller Prozess: `int init(void);`
  Erster Prozess der gestartet wird; „Vater aller Prozesse“

Beispiel (userspace/proc1.c)

```
int proc1(void)
{
  int pid_proc2;
  int exit_code_proc2;
  msg("proc1 got PID %i", get_pid());
  pid_proc2 = create_process("proc2", proc2, NULL);
  if (pid_proc2 == -1){
    msg("Failed to create proc2!");
    return -1;
  }

  if (wait_for_child(pid_proc2, &exit_code_proc2)){
    msg("Failed to wait for child proc2!");
    return -1;
  }

  msg("proc2 ended with exit code %i", exit_code_proc2);

  return 0;
}
```

Prozess-Lebenszyklus

Prozesse durchlaufen potentiell folgende Zustände:

1. Erzeugung: Prozess P wird durch `create_process` von einem anderen Prozess erzeugt; P wird als lauffähig markiert und in die `run_queue` gestellt
2. Ausführung: P wird irgendwann von `schedule` ausgewählt und ausgeführt. Die Ausführung beginnt mit Funktion `process_wrapper`, die die Hauptfunktion von P aufruft
3. Prozessumschaltung: Blockiert P (z. B. in `wait_for_process`, wird der Zustand auf `STATE_BLOCKED` gesetzt und `schedule` aufgerufen; dadurch wird P in die `wait_queue` eingereiht und ein anderer Prozess ausgeführt.
4. Beendigung: P beendet sich durch Rückkehr aus seiner Hauptfunktion, die von `process_wrapper` aufgerufen wurde (s. o.). `process_wrapper` setzt den Zustand von P auf `STATE_DEAD` und ruft `schedule` auf. In `start_scheduler` wird der Prozess dann in die `zombie_queue` aussortiert.

Signale sperren: Beispiel

SIGINT sperren

```
sigset_t sigset;
sigset_t old_set;

sigemptyset(&sigset);
sigaddset(&sigset, SIGINT);
sigprocmask(SIG_BLOCK, &sigset, &old_set);
/* Hier ist SIGINT gesperrt */

/* Sperre aufheben: */
sigprocmask(SIG_SETMASK, &old_set, NULL);
```

- Zeitgebersignale können mit `setitimer(2)` explizit angefordert werden

setitimer

```
int setitimer(int which,
              struct itimerval *value,
              struct itimerval *ovalue);


struct itimerval {
  struct timeval it_interval; /* next value */
  struct timeval it_value;    /* current value */
};
struct timeval {
  long tv_sec;                /* seconds */
  long tv_usec;               /* microseconds */
};
```

- Dekrementiert `value->it_value` bis 0 erreicht wird und schickt ein Signal (s. u.). Anschließend wird `value->it_value` wieder mit `value->it_interval` geladen
- `ovalue` kann verwendet werden um aktuellen Stand abzufragen

Zeitgeberarten

`which` wählt aus drei verschiedenen Zeitgebern

- `ITIMER_REAL`: tatsächlich vergangene Zeit, verschickt Signal `SIGALRM`
- `ITIMER_VIRTUAL`: vom Prozess verbrauchte Zeit, `SIGVTALRM`
- `ITIMER_PROF`: vom Prozess und vom BS im Auftrag des Prozesses verbrauchte Zeit, `SIGPROF`

Aufgabe 8: Vorgehensweise

Ziel: Implementierung der Round-Robin Strategie

- In der `main` Funktion Zeitgebersignal anfordern
- Zeitscheibengröße wird über **`struct`** `itimerval.it_interval` eingestellt
- Im Signalhandler (entspricht Interruptbehandlung im BS) `schedule` aufrufen
- *Überall* beim Betreten des Kerns Interrupts sperren (d. h. Zeitgebersignal sperren) und beim Verlassen freigeben
- `select_next_process` anpassen, so dass immer der erste Prozess aus der `run_queue` zurückgeliefert wird